# AN11703

## LPC5410x Sensor Processing-Motion Solution

Rev. 1.0 — 29 June 2015 Application note

**Document information**

| Info | Content |
|---|---|
| **Keywords** | LPC54100, LPC54102, Sensor Processing-Motion Solution, SPM-S, sensor hub, sensor fusion, Bosch, BSX, BSXlite, BSX Lite |
| **Abstract** | This application note describes the NXP LPC54102 Sensor Processing/Motion Solutions (SPM-S). Details on the microcontroller, its typical applications, the SPM-S sensor hub software and strategies regarding power optimization are given. |

**Revision history**

| Rev | Date | Description |
|---|---|---|
| 1.0 | 20150629 | Initial revision |

## Contact information

For more information, please visit: http://www.nxp.com

# 1. Introduction

The LPC54100 series are high performance, yet low-power dual-core ARM Cortex-M4/M0+ microcontrollers, operating at frequencies up to 100 MHz. The LPC5410x includes up to 512 kB of flash memory and up to 104 kB of SRAM. The peripheral complement includes two SPI interfaces, four USARTs, three Fast-mode Plus I$^2$C Bus interfaces (also supporting High Speed mode as a slave), a 32-bit, general purpose State Configurable Timer/PWM (SCT/PWM) and an assortment of other timers, including a Real-Time Clock module with a dedicated oscillator, a 12-channel/12-bit, 5 MSPS ADC. A DMA controller can service most of the peripherals.

Typical applications for the LPC54100 series include always-on sensor processing, as demonstrated in the NXP LPC54102 Sensor Processing-Motion Solution (SPM-S). The NXP LPC54102 SPM-S (OM13078) is a 'sensor hub solution', consisting of an LPC54102 LPCXpresso board and a sensor-shield board. The sensor shield contains multiple sensors, for example, various motion sensors (accelerometer, gyroscope, magnetometer) and other sensors such as temperature, pressure, ambient light, and proximity. The LPC54102 samples these sensors, processes the data (9 Degrees of Freedom sensor fusion using the Bosch BSXlite library) and sends the result over an I$^2$C host interface. The data can be visualized using an example interactive Windows application, which communicates to the sensor hub using the on-board I$^2$C-to-USB bridge.

This application note describes the following topics in more detail:

- Background on sensor hubs and sensor fusion.
- SPM-S system overview.
- SPM-S software overview.
- SPM-S sensor data acquisition.
- Bosch BSXlite sensor fusion.
- SPM-S host interface.

## 1.1 Sensor hubs and sensor fusion: Overview

Section 1.1.1 and section 1.1.2 give a general overview on sensor hubs and sensor fusion.

### 1.1.1 Sensor hubs

Sensor hub is one of the major applications of the LPC54100 series microcontrollers. Sensor hubs are found in mobile devices, such as smartphones, tablets, and wearables. Always-on sensors at low power consumption are made possible by sensor hubs. Before the introduction of sensor hubs, the sensors were handled by the device's application processor (AP), causing significant battery drain when these sensors would be on continuously. Sensor hubs solve this problem by performing sensor data aggregation, sensor data batching ('buffering') and sensor fusion in a low-power fashion, reducing the need for the AP to be active while the sensors are active. Fig 1 shows a typical block diagram for a sensor-hub solution.

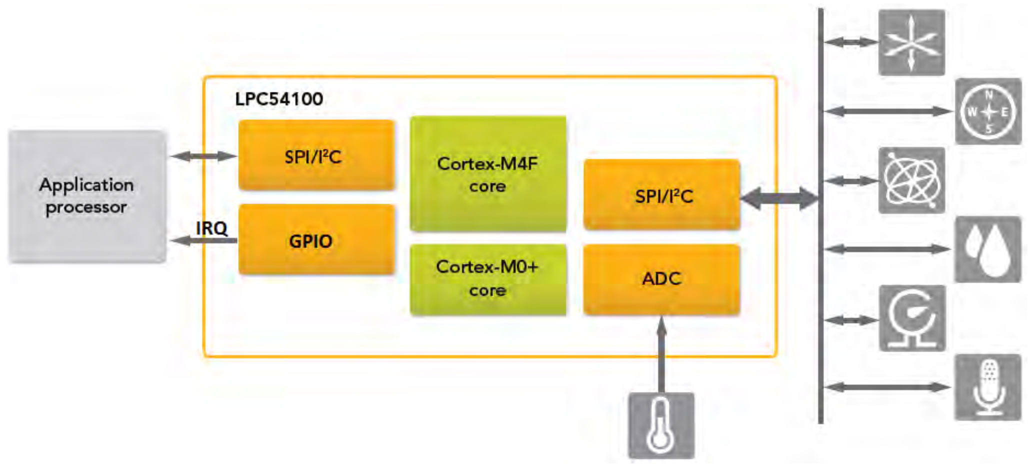


**Fig 1. Block diagram of an LPC54100-based sensor hub**

### 1.1.2 Sensor Fusion

Besides sensor sampling and sensor data batching, another major task for the sensor hub is sensor data processing, known as ‘sensor fusion’. Sensor fusion, consists of algorithms that combine the data of several sensors to get more accurate and complete sensor data. It aims to compensate shortcomings of each sensor and provides highly accurate, reliable, and stable data. Typically, sensor fusion is performed on motion sensors, such as an accelerometer, gyroscope and magnetometer, though it is also possible to perform sensor fusion with other types of sensors as input. In this document, sensor fusion refers to the 9-axis (or 9DOF, 9 Degrees of Freedom) motion sensor fusion, which fuses the data from the three above mentioned motion sensors.

Besides providing more accurate sensor data, sensor fusion can also derive easy-to-interpret data, such as the orientation of the device (for example, roll, pitch and yaw).

It is important to understand that sensor fusion algorithms rely on advanced arithmetic, which sets an important requirement for the microcontroller; it must have enough processing power to execute the sensor fusion algorithm and finish it before the next iteration is executed.

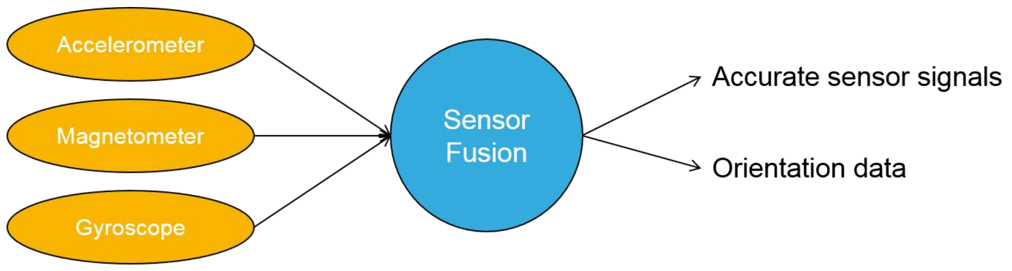


**Fig 2. Sensor fusion applied to motion sensors**

## 1.2 Sensor Processing-Motion Solution

The NXP LPC54102-based Application-in-a-box Sensor Processing-Motion Solution (OM13078) enables a new generation of always-on, context-aware products. The system listens to, monitors, and aggregates data from several sensors and processes this data using complex sensor-fusion software, included in the solution.

NXP has partnered with Bosch Sensortec to offer an integrated solution that makes it easy to incorporate motion or inertia and other sensor data into a variety of end applications. The solution includes commercial and development licenses for Bosch Sensor Fusion (BSXlite). The software combines motion-sensor data to get accurate sensor signals or derived sensory information with minimal memory requirements. It provides this motion-sensor data in the form of 9-axis motion vectors, represented as quaternions or heading, pitch and roll values.

The solution consists of an LPCXpresso54102 board and a sensor-shield board. The sensor shield contains multiple sensors, for example, various motion sensors (accelerometer, gyroscope and magnetometer) and other sensors such as temperature, pressure, ambient light, and proximity. The LPC54102 samples these sensors, processes the data, and sends the result over an I$^2$C host interface. The data is visualized using an interactive Windows application, which communicates to the sensor hub using the on-board I$^2$C-to-USB bridge.

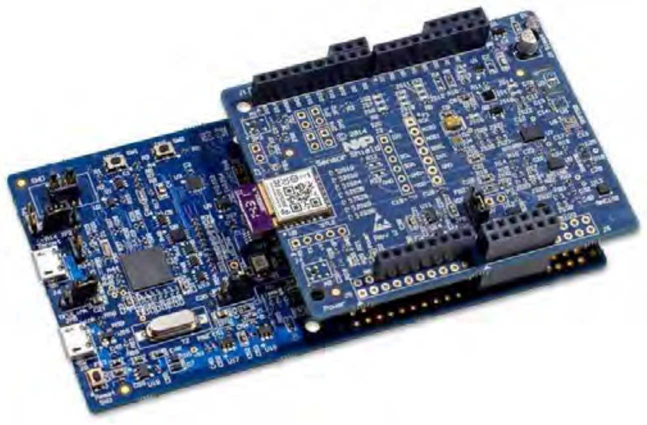
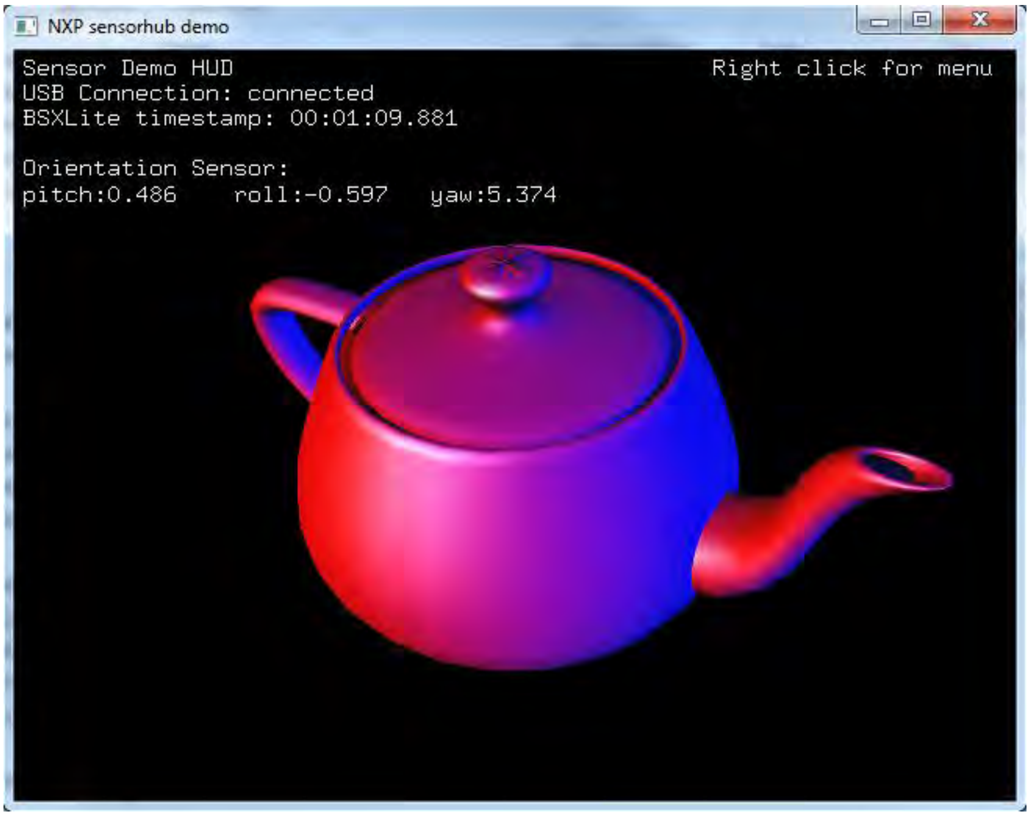


Fig 3. Sensor Processing-Motion boards (LPCXpresso54102 board with sensor shield) (OM13078)



```
Sensor Demo HUD                                    Right click for menu
USB Connection: connected
BSXLite timestamp: 00:01:09.881

Orientation Sensor:
pitch:0.486    roll:-0.597   yaw:5.374
```

Fig 4. Sample application showing a teapot, the orientation of the teapot is based on the orientation of the board

See the Sensor Processing/Motion Solution page on NXP.com for the SPM-S source code, board schematic, PC application and Quick Start Guide.

### 1.3 Typical applications

Although this application note focuses on the smartphone or tablet sensor hub use-case, there are many applications which can make use of a high-performance, low-power microcontroller, possibly combined with sensor fusion software:

- Mobile handsets and tablets.
- Portable health and fitness monitoring devices.
- Gaming devices such as 3D mice and sensor gloves.
- Head-worn glasses or terminals.
- Home and building automation products.
- Fleet management and asset tracking.
- Robotics.
- Flying drones.
- Quadcopters.

# 2. SPM-S system overview

This chapter gives a more detailed overview of the Sensor Processing/Motion Solution from a system perspective. It contains the following sections:

- Sensor hub architecture.
- SPM-S Architecture.

## 2.1 Sensor hub architecture

Fig 5 shows the block diagram of a typical sensor-hub based device.

The main components are:

- The sensor hub.
- The Application Processor (AP).
- The Host Interface (Host I/F).
- The sensors.

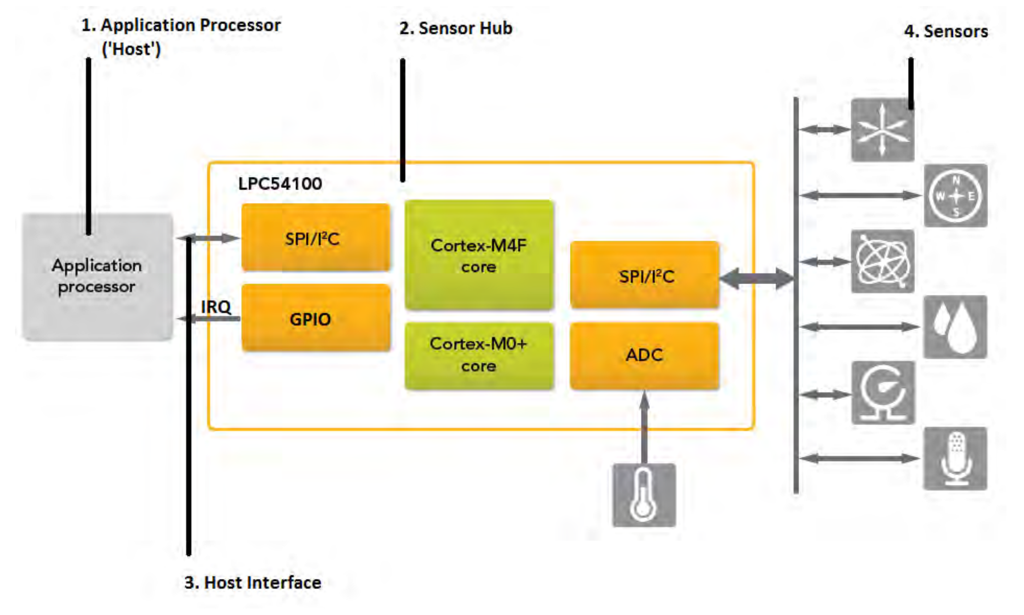


Fig 5. Architecture of a typical sensor-hub based device

### 2.1.1 Application Processor

In a typical smartphone or tablet device, the application processor can be considered as one of the main components. Among its tasks are controlling the display, controlling the radios, and running the operating system of the phone. In modern devices this is often a multi-core ARM Cortex-A SoC, running at high frequencies (>1 GHz). Application processors have a sleep mode, which is entered as often as possible to save power.

A sensor hub is added to off-load the so-called 'always-on' tasks from the AP, such as sensor interfacing, to lower the power consumption of the device. The AP determines which sensors should be enabled and at which update rate, and informs the sensor hub about this. When the sensor hub has gathered the sensor data, it wakes up the AP and transfers the data.

### 2.1.2 Sensor hub

The sensor hub performs tasks which require less processing power that should be running while the device is in a low-power mode (AP in sleep mode). Commands are received from the AP, telling the sensor hub which sensors should be enabled and under which circumstances the sensor hub can wake up the AP. The sensor hub samples the sensors and stores the sensor data, wakes up the AP when the wake-up condition is met, and transfers the data to the AP.

### 2.1.3 Host interface

The host interface is the communication between the host (the AP) and the sensor hub. Most often, the host interface is an $I^2C$ or SPI bus. The host sends commands to the sensor hub, and the sensor hub can transfer responses and sensor data to the host.

Since the sensor hub is a slave controlled by the host, an IRQ signal is added so that the sensor hub can notify the host that it wants to send a response or sensor data.

### 2.1.4 Sensors

The sensors deliver the raw data for the sensor hub. Typically these sensors are connected to the sensor hub using an $I^2C$ bus, though they can also be interfaced through SPI, or even using an ADC when dealing with analog sensors.

## 2.2 SPM-S Architecture

The SPM-S architecture can be divided into hardware architecture and a software architecture. Section 2.2.1 and section 2.2.2 describes the architecture and the important difference between the ‘virtual sensors’ and ‘physical sensors’.

### 2.2.1 Hardware Architecture

The SPM-S consists of two boards; the LPCXpresso54102 board which includes the LPC54102 and an on-board debugger, and the Sensor Shield Board, which includes the motion and environmental sensors. See Fig 6.

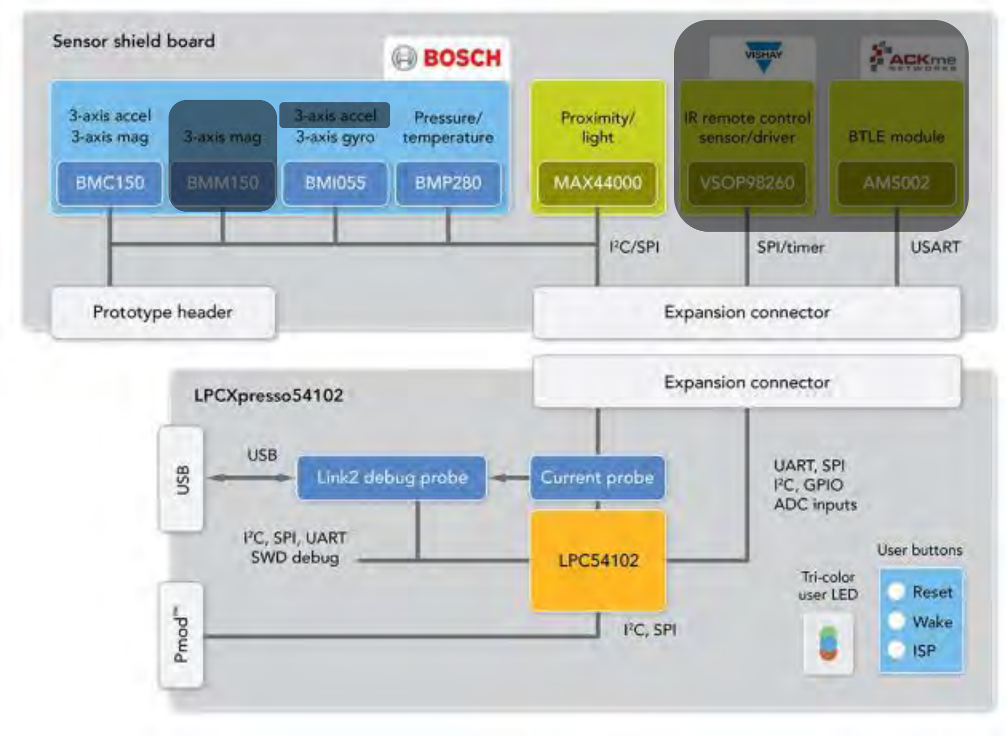


Fig 6. SPM-S hardware block diagram. Shaded components not used by the demo

Note that some of the motion sensors on the board are (partially) redundant as one of their functions is also implemented by another sensor. As an example, for the accelerometer, both the BMC150 and the BMI055 can be used. This overlapping sensor functionality is provided to enable different performance to be achieved at the system level; for example, a high end mobile phone would use a BMI055 inertial sensor (with accelerometer) and a separate magnetometer (BMM150), whereas, a low end phone may only use an electronic compass (BMC150) and emulate a gyro. The shaded areas in Fig 6 indicate this sensor is not being used by the demo. Also, the IR remote control and the Bluetooth Low Energy module are not used in the SPM-S demo.

All sensors are connected to the LPC54102 using a single I²C bus. Table 1 shows the sensors that are used by the demo, the I²C address at which they are accessible and the software driver that implements it. (See section 3.3 for Source code and directory structure).

Table 1. Sensors used by the SPM-S demo

| Sensor | Function | 7-bit I$^2$C address | Driver |
|---|---|---|---|
| BMC150 | 3-axis accelerometer | 0x10 (b0010000) | bma2x2.c |
| | 3-axis magnetometer | 0x12 (b0010010) | bmm050.c |
| BMI055 | 3-axis gyroscope[1] | 0x69 (b1101001) | bmg160.c |
| BMP280 | Pressure | 0x76 (b1110110) | bmp280.c |
| | Temperature | 0x76 (b1110110) | bmp280.c |
| MAX44000 | Proximity | 0x4A (b1001010) | max44000.c |
| | Light | 0x4A (b1001010) | max44000.c |

The host interface is implemented using a second I$^2$C bus, which is connected to the on-board debugger. This on-board debug probe also implements an I$^2$C-to-USB bridge, allowing the sensor-hub's I$^2$C host interface to be easily connected to a PC.

## 2.2.2 Software Architecture

NXP has developed a generic 'LPC Sensor Framework', which forms the main part of the SPM-S. See Fig 7. The framework relies on the Bosch sensor driver to interface with the sensors and the LPCOpen Driver libraries to communicate with all the on-chip peripherals.

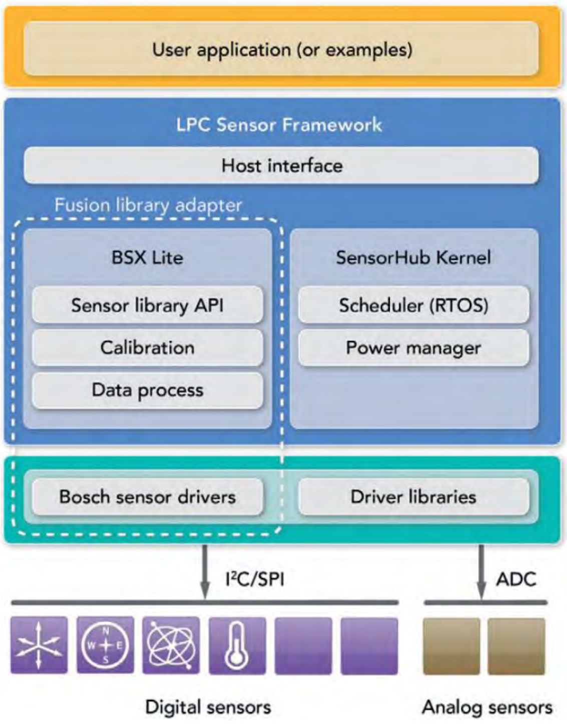


Fig 7. SPM-S software block diagram

1. [1] BMI055 is a 6-axis inertial sensor, consisting of an accelerometer and a gyroscope. In the demo software we only use its gyroscope function and use the BMC150 for the accelerometer data.

The LPC Sensor Framework encapsulates:

- A generic sensor driver module, to abstract all the different sensor drivers and create a uniform sensor API.
- A host interface module, which allows to receive commands, buffer data and responses, and transmit the buffered responses or data over the physical host interface.
- A timer module to allow periodic polling of sensors and sensor time-stamping.
- A power manager to dynamically adjust the CPU clock speed, enter low-power modes and enable/disable peripherals.
- An abstraction layer to adopt the LPC Sensor Framework to the Bosch BSXlite library.
- The Bosch BSXlite library (in object format).
- (optionally) freeRTOS to take care of executing multiple tasks in parallel.

### 2.2.3 Virtual Sensors and Physical Sensors

It is important to note that the SPM-S makes a distinction between virtual sensors and physical sensors.

Physical sensors are the physical devices connected as input to the sensor hub. For example, an accelerometer, gyroscope, or proximity sensor.

Virtual sensors are the sensors and their data as perceived by the host. For example, an ‘orientation sensor’, a calibrated accelerometer, or a calibrated gyroscope. Virtual sensors can consist of multiple physical sensors, where sensor fusion is used to obtain the sensor data, but they can also consist of a single sensor where the virtual sensor data is the raw data from the physical sensor.

The SPM-S must be able to map the virtual sensors to one or more physical sensors because the host tells the sensor hub to enable a certain virtual sensor and then expects to receive the data of that virtual sensor.

See section 4.3. Tying-in the sensor for a description of the process.

# 3. SPM-S Software Overview

This chapter describes the embedded software for the SPM-S. It contains the following sections:

- Software flow.
- Software stack.
- Source code and directory structure.

## 3.1 Software flow

Fig 8 shows a simplified software flow chart of the SPM-S firmware.

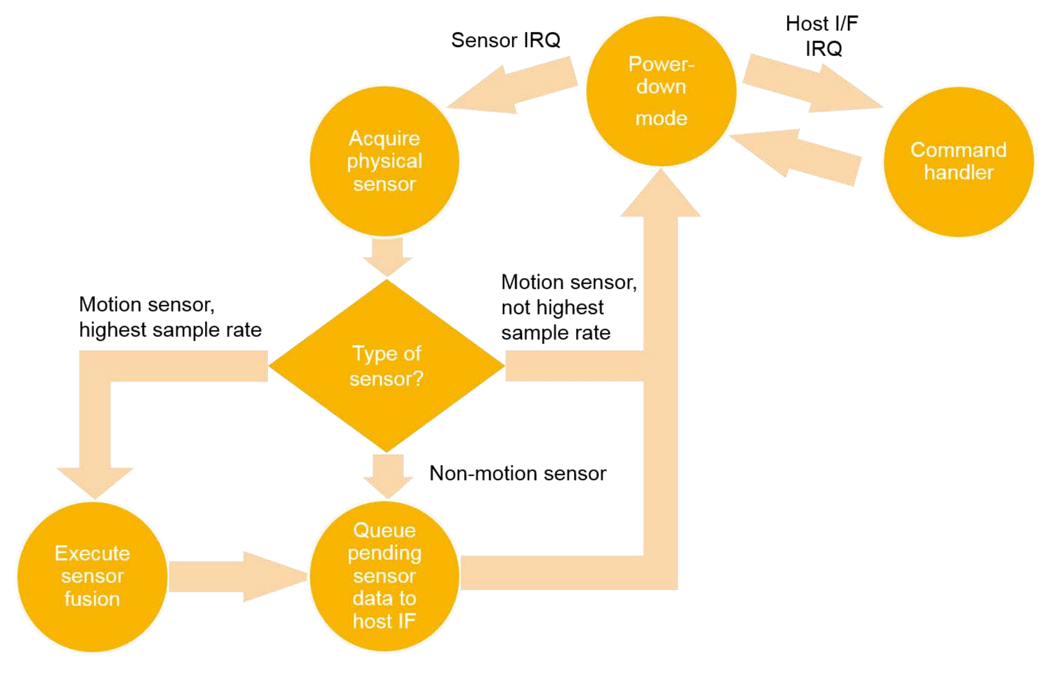


**Fig 8. Simplified software flow diagram of the SPM-S firmware**

The software flow is as follows:

- To achieve the lowest power consumption, the LPC54102 remains in low-power mode when it is idle.
- The LPC54102 can be woken up by either the host interface (reception of command), or by a trigger indicating that one or more sensors must be sampled.
- If woken up by the host interface, the LPC54102 receives the command and its parameters (section 6.3.1. Commands) and the software executes the command accordingly (For example, enable/disable specific sensors, change the sample rate). When done, it goes back into low-power mode again.
- If woken up by a sensor trigger, the software acquires and stores the sample first. Depending on the sensor type (For example, accelerometer, gyroscope, or temperature.), the software waits for more sensor data, performs the sensor fusion,

or queues the data to be sent over the host interface. When done, the LPC54102 goes back into low-power mode again.

## 3.2 Software stack

Fig 9 shows a more complete software stack.

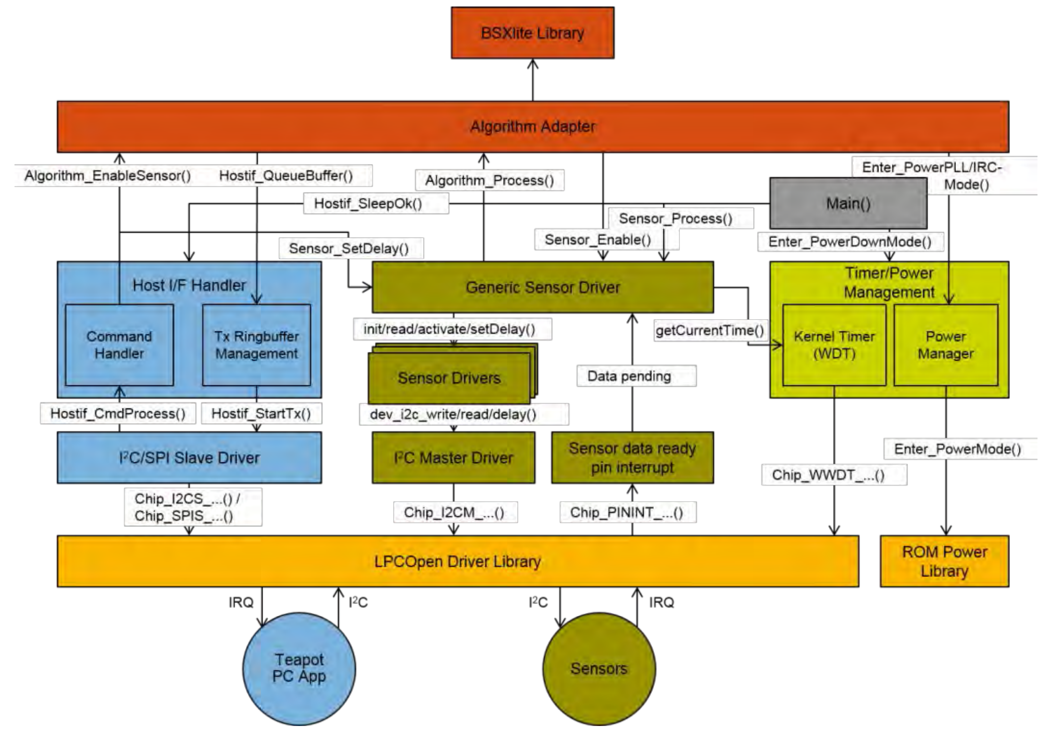


**Fig 9. SPM-S software stack**

The following functional blocks are shown in Fig 9:

- **Host & Host interface (in blue)**
  The host can send commands to the sensor hub through its I²C interface. The LPCOpen I²C slave driver is used to receive the commands and to pass them on to the *command handler*. Responses and sensor data to be sent to the host are queued in a ring buffer using the *Tx Ringbuffer Management*. The queued data is transferred over I²C using the LPCOpen I²C slave driver.
  The workings of the host interface is described in details in section 6. SPM-S Host Interface.
- **The sensors and sensor drivers (in dark green)**
  Sensors can either be polled or are interrupt driven. In case of an interrupt driven sensor, the sensor asserts its IRQ signal when a new sample is available and the LPC54102 executes the corresponding *PININT* interrupt handler. This interrupt handler only saves the time-stamp and increase a *data pending* counter.
  *Main()* periodically executes *Sensor_Process()* (located in the *generic sensor driver*) to check for new samples (*data pending* counter for IRQ-based sensors, *current time* for polled sensors) and reads the samples if available. Samples are read using their respective *sensor drivers*. Samples are then passed on to the

*algorithm adapter* to further process the sample, and/or to queue them in the *host interface handler's* Tx ring buffer.

- Section 4 SPM-S Sensor Data Acquisition describes the sampling of sensors and processing of the samples.
- **The kernel timer and power management (in light green)**
  The *kernel timer* provides functions to easily get the current time (for example, used for timestamping), while the *power management* module provides functions to easily switch between power-down mode, low-power low-performance mode (CPU @12MHz) and high-performance mode (CPU @84MHz).
- **Bosch BSXlite & BSXlite adapter (in red)**
  The *BSXlite library* performs the actual sensor fusion, while the *algorithm adapter* is an abstraction layer to connect the BSXlite to the LPC Sensorhub Framework.
- **LPCOpen and ROM peripheral driver (in orange)**
  The *LPCOpen driver library* and the *ROM power library* provide access to the LPC54102's peripherals and ROM drivers.

## 3.3 Source code and directory structure

The SPM-S source code is released in LPCOpen V2 format for LPCXpresso, Keil, and IAR. Fig 10 shows a high-level structure of the source code for LPCXpresso.

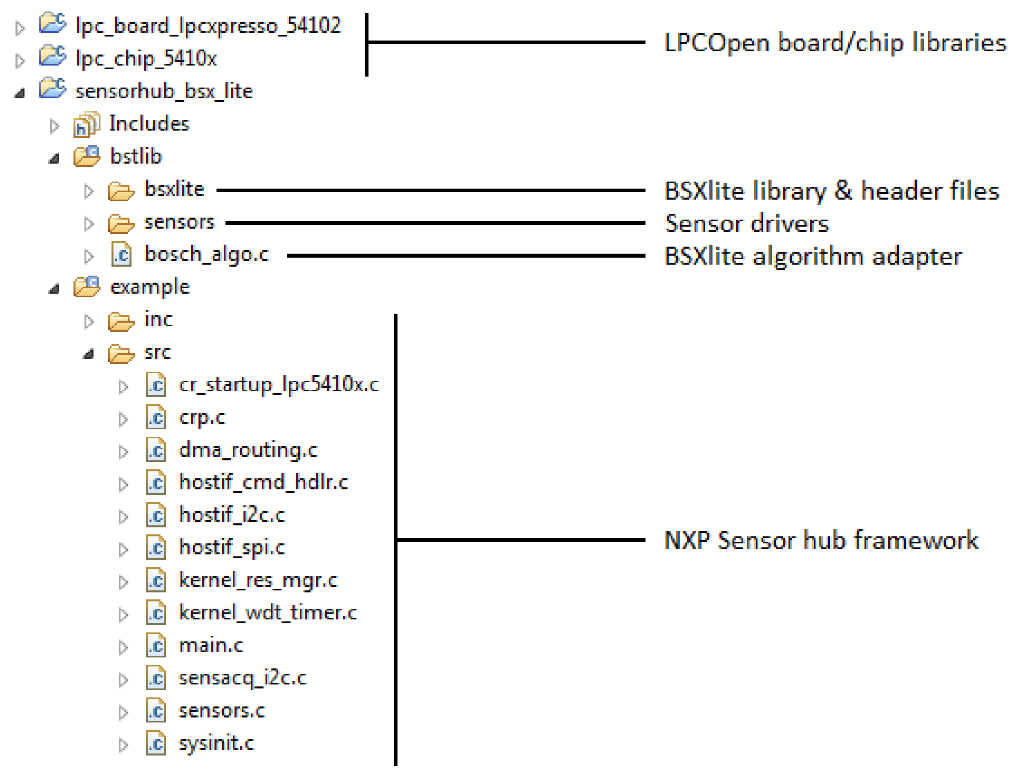


Fig 10. Source code directory structure

# 4. SPM-S Sensor Data Acquisition

One of the main tasks for the LPC Sensor Framework is sampling the sensors. A generic sensor software-interface has been designed to be able to support different kinds of sensors while maintaining the same API. This chapter explains how sensor sampling is handled, what the generic interface looks like and how it is (logically) connected to the virtual sensors that can be enabled by the host.

## 4.1 Basics

As briefly explained in section 3.2. Software stack, the SPM-S supports two kinds of sensors:

1. Sensors that can autonomously sample at a certain sample rate and generate interrupts when a new sample is available.
2. Sensors that must be instructed to take a sample.

With each sample, the timestamp of the sample is also taken, so that the sensor fusion and the host know when the sample was taken, which may be required for certain algorithms.

Interrupt driven sensors have their own IRQ handler that gets executed upon assertion of the IRQ signal using the LPC54102 PININT peripheral. This IRQ handler only stores a timestamp and increases a global ‘irq pending’ variable; the actual sample of the sensor is not being read in the IRQ handler.

The actual reading of the sample is performed from *main()*; upon wake-up (and after servicing the wake-up source, for example, the sensor IRQ handler), *main()* will call function *Sensor_process()*. For interrupt driven sensors, *Sensor_process()* will read the sensor when its ‘irq pending’ counter indicates that at least one new sample is available. For polled sensors, the delta between the current timestamp and the last sample timestamp of that sensor is compared against the sample interval of that sensor. If this interval has been exceeded, the sensor will be sampled.

After reading a sample from a sensor, *Sensor_process()* calls *Algorithm_Process()* function for that sensor, which processes the raw sample and stores the processed data in the host interface Tx queue.

Fig 11 shows a simplified schematic representation of the sensor sampling process flow.

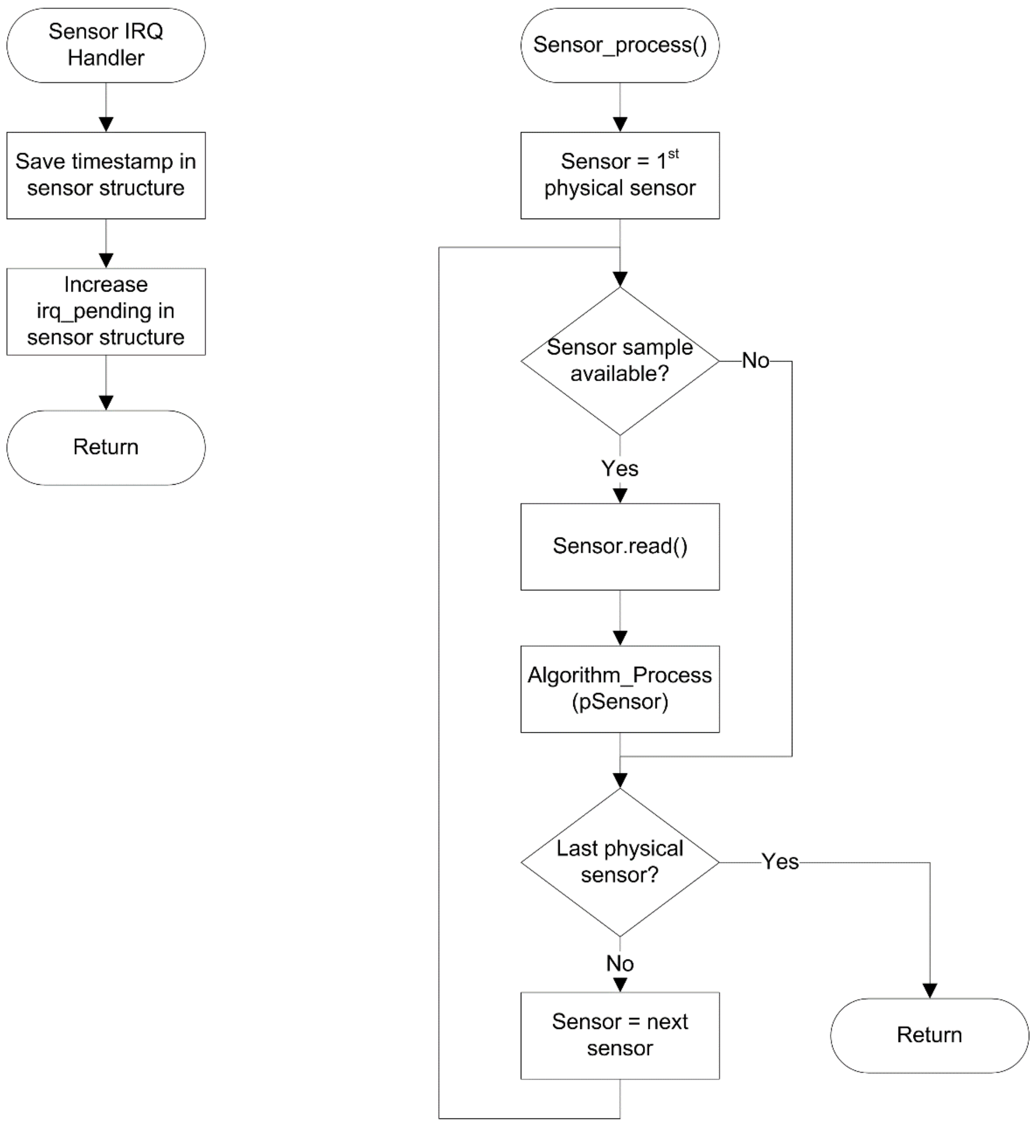


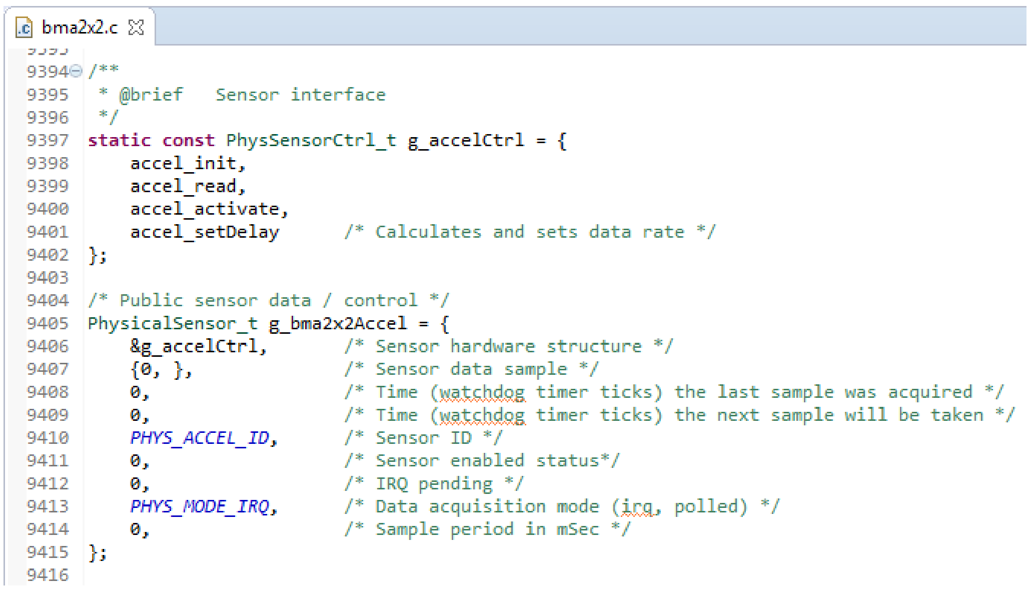
**Fig 11. Simplified flow chart of the sensor sampling process**

## 4.2 Generic sensor driver interface

The generic sensor driver interface defines a basic API that allows easy addition of different kinds of physical sensors and an easy way to map those to virtual sensors. This is accomplished by providing a structure with the sensor driver, which contains:

- Unique ID of the physical sensor.
- A table pointing to four functions, to initialize the sensor, read a sample from the sensor, to enable/disable the sensor, and to configure the sample rate of the sensor.
- 64 bits to store sample data.

- Timestamp of last sample and when the next sample should be taken.
- Flag to indicate the sensor is enabled or disabled.
- The ‘irq pending’ counter.
- The sensor’s mode (IRQ driven of polled).
- The sensor’s configured sample rate.



```
/**
 * @brief   Sensor interface
 */
static const PhysSensorCtrl_t g_accelCtrl = {
    accel_init,
    accel_read,
    accel_activate,
    accel_setDelay      /* Calculates and sets data rate */
};

/* Public sensor data / control */
PhysicalSensor_t g_bma2x2Accel = {
    &g_accelCtrl,       /* Sensor hardware structure */
    {0, },              /* Sensor data sample */
    0,                  /* Time (watchdog timer ticks) the last sample was acquired */
    0,                  /* Time (watchdog timer ticks) the next sample will be taken */
    PHYS_ACCEL_ID,      /* Sensor ID */
    0,                  /* Sensor enabled status*/
    0,                  /* IRQ pending */
    PHYS_MODE_IRQ,      /* Data acquisition mode (irq, polled) */
    0,                  /* Sample period in mSec */
};
```

**Fig 12. The physical sensor structure and function-pointer table for the Bosch accelerometer. Driver is applicable to both BMC150 and BMI055, BMC150 is selected by I$^2$C slave address by demo software**

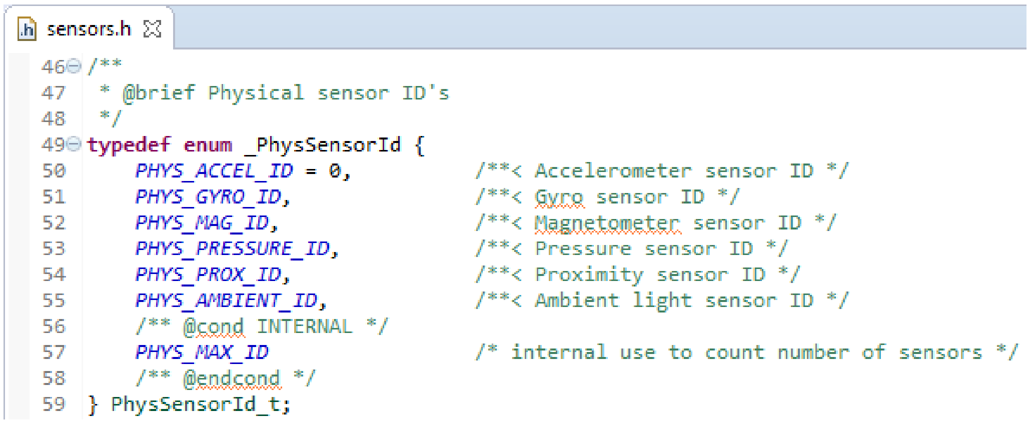
## 4.3 Tying-in the sensor driver

The generic sensor driver interface helps to keep the sensor management simple, flexible, and easy to expand but does not do anything by itself. Each sensor must be connected to the rest of the framework so that:

- It has a unique physical sensor ID to be able to identify the sensor.
- The sensor is included in the list of physical sensors, so that the sensor is sampled on the specified interval.
- It belongs to a virtual sensor (enabling of virtual sensor should eventually result in the enabling of one or more physical sensors).
- The sensor data is either further processed and/or stored in the host I/F Tx queue.

### 4.3.1 Unique physical sensor ID

Each physical sensor can be identified by its unique sensor ID. These IDs are defined using the *PhysSensorId_t* enumerator (sensors.h). See Fig 13.



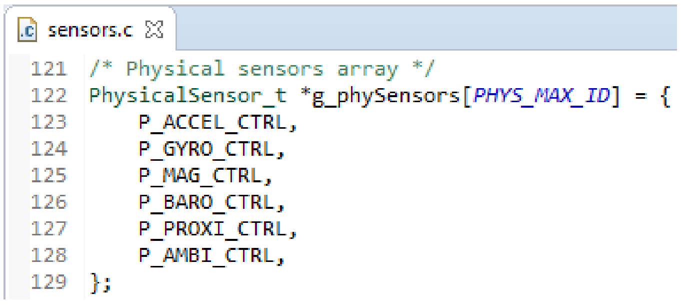
sensors.h

```
/**
 * @brief Physical sensor ID's
 */
typedef enum _PhysSensorId {
    PHYS_ACCEL_ID = 0,          /**< Accelerometer sensor ID */
    PHYS_GYRO_ID,               /**< Gyro sensor ID */
    PHYS_MAG_ID,                /**< Magnetometer sensor ID */
    PHYS_PRESSURE_ID,           /**< Pressure sensor ID */
    PHYS_PROX_ID,               /**< Proximity sensor ID */
    PHYS_AMBIENT_ID,            /**< Ambient light sensor ID */
    /** @cond INTERNAL */
    PHYS_MAX_ID                 /* internal use to count number of sensors */
    /** @endcond */
} PhysSensorId_t;
```

Fig 13. Unique sensor ID of each physical sensor

### 4.3.2 g_phySensors array

The g_phySensors array (sensors.c) holds pointers to all physical-sensor data/control structures (*PhysicalSensor_t* structures as shown in Fig 12). This array is traversed when a sensor is enabled with a specific ID, or by the *Sensor_process()* function to check whether a new sample is available.



sensors.c

```
/* Physical sensors array */
PhysicalSensor_t *g_phySensors[PHYS_MAX_ID] = {
    P_ACCEL_CTRL,
    P_GYRO_CTRL,
    P_MAG_CTRL,
    P_BARO_CTRL,
    P_PROXI_CTRL,
    P_AMBI_CTRL,
};
```

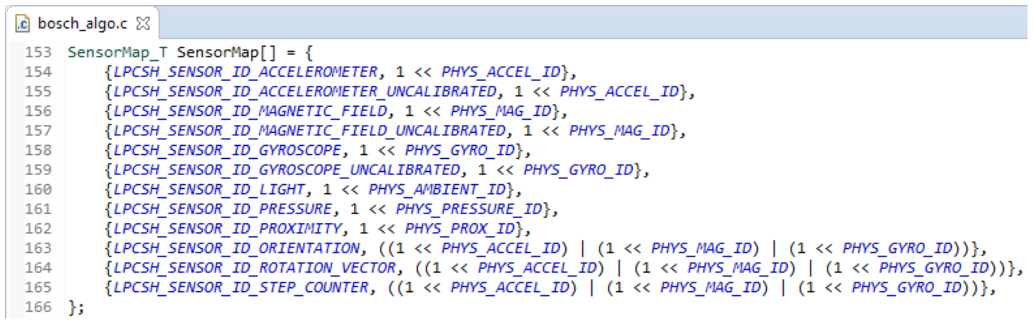
**Note:** Each entry in the displayed table is a pointer to a *PhysicalSensor_t* structure like the one shown in Fig 12 using *#define* macros

Fig 14. Array g_phySensors

### 4.3.3 SensorMap array

The SensorMap array (bosch_algo.c) maps virtual sensors to one or more physical sensors. Each virtual sensor has a single entry in this array, specifying which physical sensors are being used by that virtual sensor.

The SensorMap array is used to enable/disable the right physical sensors when the host enables/disables a certain virtual sensor. In Fig 15, the virtual 'orientation sensor' uses the accelerometer, magnetometer, and gyroscope as an example.



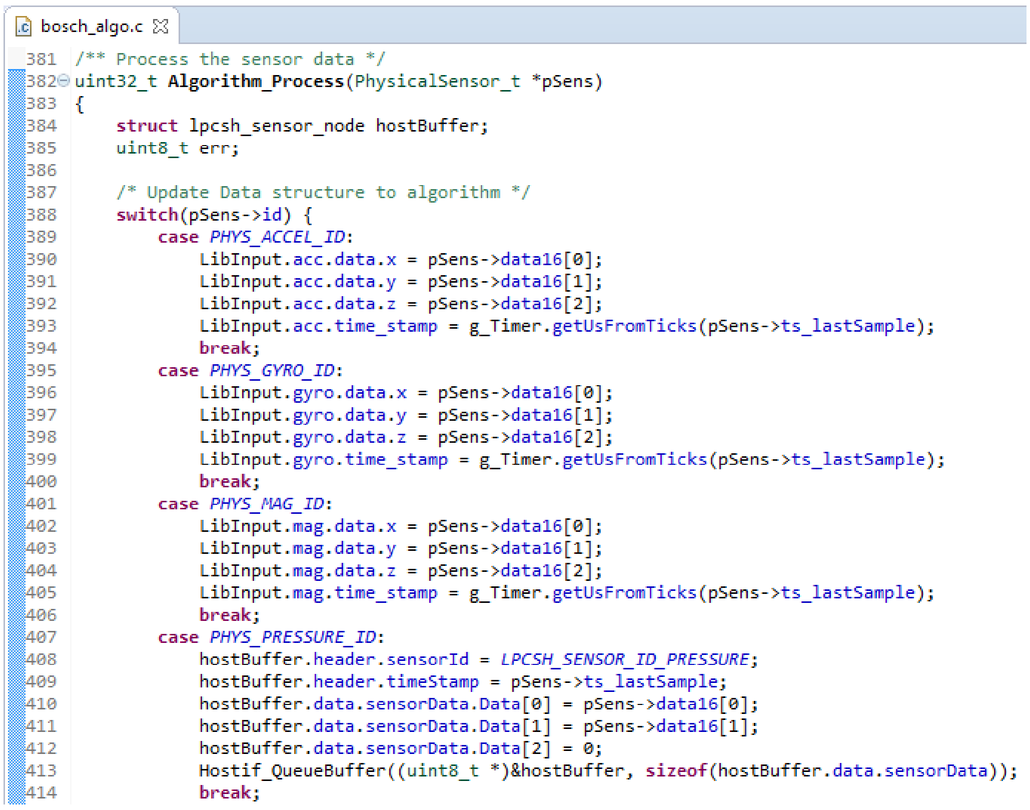
bosch_algo.c

```
SensorMap_T SensorMap[] = {
    {LPCSH_SENSOR_ID_ACCELEROMETER, 1 << PHYS_ACCEL_ID},
    {LPCSH_SENSOR_ID_ACCELEROMETER_UNCALIBRATED, 1 << PHYS_ACCEL_ID},
    {LPCSH_SENSOR_ID_MAGNETIC_FIELD, 1 << PHYS_MAG_ID},
    {LPCSH_SENSOR_ID_MAGNETIC_FIELD_UNCALIBRATED, 1 << PHYS_MAG_ID},
    {LPCSH_SENSOR_ID_GYROSCOPE, 1 << PHYS_GYRO_ID},
    {LPCSH_SENSOR_ID_GYROSCOPE_UNCALIBRATED, 1 << PHYS_GYRO_ID},
    {LPCSH_SENSOR_ID_LIGHT, 1 << PHYS_AMBIENT_ID},
    {LPCSH_SENSOR_ID_PRESSURE, 1 << PHYS_PRESSURE_ID},
    {LPCSH_SENSOR_ID_PROXIMITY, 1 << PHYS_PROX_ID},
    {LPCSH_SENSOR_ID_ORIENTATION, ((1 << PHYS_ACCEL_ID) | (1 << PHYS_MAG_ID) | (1 << PHYS_GYRO_ID))},
    {LPCSH_SENSOR_ID_ROTATION_VECTOR, ((1 << PHYS_ACCEL_ID) | (1 << PHYS_MAG_ID) | (1 << PHYS_GYRO_ID))},
    {LPCSH_SENSOR_ID_STEP_COUNTER, ((1 << PHYS_ACCEL_ID) | (1 << PHYS_MAG_ID) | (1 << PHYS_GYRO_ID))},
};
```

**Fig 15. The SensorMap array defines which physical sensors are being used by a specific virtual sensor**

### 4.3.4 Algorithm_Process() function

The function *Algorithm_Process()* takes care of processing the raw sensor data and queueing it in the host I/F Tx buffer. Therefore, each physical sensor must have an entry in the function's *switch/case* statement. For some sensors, such as a temperature sensor or a proximity sensor, this can be easy and only requires pushing the data to the queue. For other sensors, such as the motion sensors, it is more complicated because the data must be processed first.



bosch_algo.c

```
/** Process the sensor data */
uint32_t Algorithm_Process(PhysicalSensor_t *pSens)
{
    struct lpcsh_sensor_node hostBuffer;
    uint8_t err;

    /* Update Data structure to algorithm */
    switch(pSens->id) {
        case PHYS_ACCEL_ID:
            LibInput.acc.data.x = pSens->data16[0];
            LibInput.acc.data.y = pSens->data16[1];
            LibInput.acc.data.z = pSens->data16[2];
            LibInput.acc.time_stamp = g_Timer.getUsFromTicks(pSens->ts_lastSample);
            break;
        case PHYS_GYRO_ID:
            LibInput.gyro.data.x = pSens->data16[0];
            LibInput.gyro.data.y = pSens->data16[1];
            LibInput.gyro.data.z = pSens->data16[2];
            LibInput.gyro.time_stamp = g_Timer.getUsFromTicks(pSens->ts_lastSample);
            break;
        case PHYS_MAG_ID:
            LibInput.mag.data.x = pSens->data16[0];
            LibInput.mag.data.y = pSens->data16[1];
            LibInput.mag.data.z = pSens->data16[2];
            LibInput.mag.time_stamp = g_Timer.getUsFromTicks(pSens->ts_lastSample);
            break;
        case PHYS_PRESSURE_ID:
            hostBuffer.header.sensorId = LPCSH_SENSOR_ID_PRESSURE;
            hostBuffer.header.timeStamp = pSens->ts_lastSample;
            hostBuffer.data.sensorData.Data[0] = pSens->data16[0];
            hostBuffer.data.sensorData.Data[1] = pSens->data16[1];
            hostBuffer.data.sensorData.Data[2] = 0;
            Hostif_QueueBuffer((uint8_t *)&hostBuffer, sizeof(hostBuffer.data.sensorData));
            break;
```

**Fig 16. Code snippet of the *Algorithm_Process()* function**

# 5. Bosch BSXlite Sensor Fusion

## 5.1 BSXlite overview

BSXlite is motion sensor-fusion library, provided under a limited-use Software License Agreement, which can be used for free on NXP LPC microcontrollers to process data from Bosch Sensortec sensors. BSXlite allows high-efficiency, high-performance 9-axis sensor fusion.

## 5.2 BSXlite vs. BSX

BSXlite is a feature reduced and less optimized version of the Bosch BSX library. To license the full BSX library, contact Bosch Sensortec directly. The differences between BSX and BSXlite are explained in Table 2 and Table 3.

**Table 2. Overview of differences between BSX and BSXlite**

| | **BSXlite** | **BSX (full library)** |
|---|---|---|
| Release format | Closed source code / compiled library | Closed source code / compiled library |
| License | Click-through on LPCWare.com | Contact Bosch Sensortec |
| Support / Maintenance | Limited (via LPCWare.com forums) | Full |
| **Key Features** | **BSXlite** | **BSX (full library)** |
| Axis remapping | ✗ (must be implemented outside library) | ✓ |
| Offset correction | ✓ | ✓ |
| Soft Iron Correction | ✗ (can be implemented outside library) | ✓ |
| Accelerometer calibration | ✗ | ✓ |
| Magnetometer calibration | Classic: based on figure-of-eight motion | Classic and advanced (fast calibration) |
| Magnetic distortion check | Basic | Advanced |
| Gyroscope calibration | ✓ | ✓ |
| 9-axis orientation processing | Basic | Advanced |
| Compass orientation processing | Basic (tilt compensation) | Advanced (adaptive filtering, tilt compensation) |
| Data fusion mode | 9-axis | 9-axis & 6-axis (IMU, M4G, eCompass) |
| **System Requirements** | **BSXlite** | **BSX (full library)** |
| ROM | 52k | 67k |
| RAM | 2k | 7k |

Table 3. Differences between BSX and BSXlite with regard to library output

| Outputs | BSXlite | BSX (full library) |
|---|---|---|
| Acceleration | Raw | ✓ |
| Magnetometer | Raw, corrected | ✓ |
| Gyroscope | Raw, corrected | ✓ |
| Virtual gyroscope (M4G) | × | ✓ |
| Quaternions | ✓ | ✓ |
| Orientation | ✓ (unfiltered) | ✓ |
| Rotation matrix | × | ✓ |
| Heading accuracy | ✓ | ✓ |
| Linear acceleration | × | ✓ |
| Gravity | × | ✓ |
| Gestures | × | ✓ |
| Step counter and step detector | × | ✓ |
| Significant motion | × | ✓ |
| **Output data rates (ODR)** | **BSXlite** | **BSX (full library)** |
| Accelerometer | 100 Hz | Multiple data rates |
| Magnetometer | 25 Hz | Multiple data rates |
| Gyroscope | 100Hz | Multiple data rates |
| Orientation sensor | 50 Hz | Multiple data rates |

## 5.3 Using BSXlite

The following section explains which functions are available when using BSXlite and how BSXlite should be used. More detailed information is available in the various Bosch BSXlite documents available on LPCWare.com.

### 5.3.1 BSXlite initialization

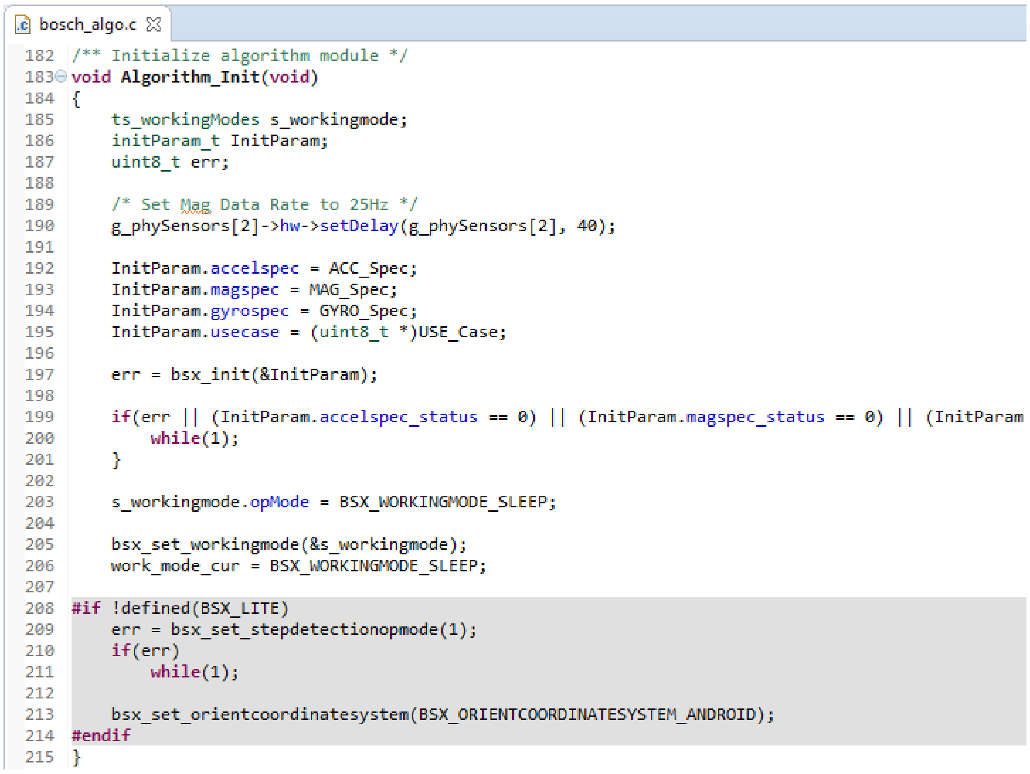
At startup, the library first must be initialized. The first step in this process is to call function *bsx_init()*. The function takes one parameter, which is a configuration structure. The inputs for this configuration structure are:

- BSX_U8 *accelspec. Contains acceleration sensor related settings for the BSX library. Specs for several Bosch accelerometers (BMA255, BMI160, BMA250, and BMA280) are included in the SPM-S software.
- BSX_U8 *magspec. Contains magnetometer sensor related settings for the BSX library. Specs for several Bosch magnetometers (BMM150, BMM050) are included in the SPM-S software.
- BSX_U8 *gyrospec. Contains gyroscope related settings for the BSX library. A spec for the Bosch BMG160 gyroscope is included in the SPM-S software.

- BSX_U8 *usecase. Contains additional required configuration of various internal modules of the library. A valid configuration for BSXlite has been included in the SPM-S software.

See the BSXlite documentation for more information on initialization and the accel/mag/gyro spec and usecase arrays.

The next step is to configure the BSX *working mode*. For BSXlite, the working mode can either be *BSX_WORKINGMODE_SLEEP* (when no sensor fusion is performed) or *BSX_WORKINGMODE_NDOF* when performing 9-axis sensor fusion. The working mode can be configured by calling function *bsx_set_workingmode()*.



bosch_algo.c

```
/** Initialize algorithm module */
void Algorithm_Init(void)
{
    ts_workingModes s_workingmode;
    initParam_t InitParam;
    uint8_t err;

    /* Set Mag Data Rate to 25Hz */
    g_phySensors[2]->hw->setDelay(g_phySensors[2], 40);

    InitParam.accelspec = ACC_Spec;
    InitParam.magspec = MAG_Spec;
    InitParam.gyrospec = GYRO_Spec;
    InitParam.usecase = (uint8_t *)USE_Case;

    err = bsx_init(&InitParam);

    if(err || (InitParam.accelspec_status == 0) || (InitParam.magspec_status == 0) || (InitParam
        while(1);
    }

    s_workingmode.opMode = BSX_WORKINGMODE_SLEEP;

    bsx_set_workingmode(&s_workingmode);
    work_mode_cur = BSX_WORKINGMODE_SLEEP;

#if !defined(BSX_LITE)
    err = bsx_set_stepdetectionopmode(1);
    if(err)
        while(1);

    bsx_set_orientcoordinatesystem(BSX_ORIENTCOORDINATESYSTEM_ANDROID);
#endif
}
```

**Fig 17. Code snippet of BSXlite initialization**

During initialization the working mode is set to *BSX_WORKINGMODE_SLEEP*. Whenever the host enables any of the BSXlite virtual sensors (*orientation* or *rotation vector*), the mode is changed to *BSX_WORKINGMODE_NDOF.*

### 5.3.2 Executing the sensor fusion

The BSXlite library can be instructed to run the sensor fusion algorithm by calling *bsx_dostep()*. *bsx_dostep()* takes a single parameter, a pointer to a *libraryinput_t* structure. *libraryinput_t* holds the data and timestamps for the accelerometer, magnetometer, and gyroscope. The sensor data should be in 32-bit signed format and the 64-bit timestamp in microseconds. The function returns a 0 when done with no errors.

### 5.3.3 Get outputs from the library

Data from the enabled virtual sensors can be read from the BSXlite library after bsx_*dostep()* has returned successfully using the functions mentioned in Table 4.

Table 4. BSXlite output 'get' functions

| Virtual sensor | Function | Output unit |
|---|---|---|
| Accelerometer | bsx_get_accrawdata() | m/s$^2$ |
| Magnetometer | bsx_get_magrawdata() | μT |
| | bsx_get_magcordata() | μT |
| Gyroscope | bsx_get_gyrorawdata_rps() | °/s |
| | bsx_get_gyrocordata_rps() | °/s |
| Rotation vector | bsx_get_orientdata_quat() | Quaternion, W, X, Y and Z as 32-bit Q24 fixed point integers |
| | bsx_get_orientdata_euler_rad() | Radians |
| Geomagnetic rotation vector | bsx_get_georotationvector_quat() | Quaternion, W, X, Y and Z as 32-bit Q24 fixed point integers |
| | bsx_get_geoheadingaccuracy_rad() | Radians |

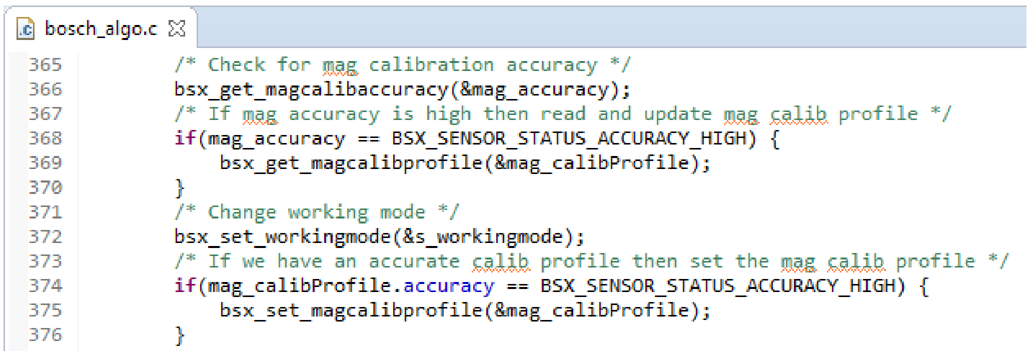
### 5.3.4 Calibration profiles

BSXlite allows saving the current magnetometer calibration data, which can be used to obtain a quicker geomagnetic fix when re-enabling the magnetometer.

To use this feature, follow below steps when changing the BSX workingmode:

- Call function bsx_get_magcalibaccuracy() to get the current magnetometer accuracy. When it's equal to BSX_SENSOR_STATUS_ACCURACY_HIGH, save the current profile to the LPC54102 SRAM using bsx_get_magcalibprofile()
- Change the working mode using bsx_set_workingmode()
- If the saved profile indicates a high accuracy, restore the calibration profile using bsx_set_magcalibprofile()

Fig 18 lists the code that shows the algorithm.



```c
        /* Check for mag calibration accuracy */
        bsx_get_magcalibaccuracy(&mag_accuracy);
        /* If mag accuracy is high then read and update mag calib profile */
        if(mag_accuracy == BSX_SENSOR_STATUS_ACCURACY_HIGH) {
            bsx_get_magcalibprofile(&mag_calibProfile);
        }
        /* Change working mode */
        bsx_set_workingmode(&s_workingmode);
        /* If we have an accurate calib profile then set the mag calib profile */
        if(mag_calibProfile.accuracy == BSX_SENSOR_STATUS_ACCURACY_HIGH) {
            bsx_set_magcalibprofile(&mag_calibProfile);
        }
```

**Fig 18. Code listing for (re)storing the magnetometer calibration profile**

# 6. SPM-S Host Interface

## 6.1 Basics

'Host interface' is the generic term for the interface between the host and the sensor hub. It allows the host to send commands to the sensor hub, and the sensor hub to send sensor data to the host.

## 6.2 Physical layer

In case of the SPM-S, the physical layer of the host interface is $I^2C$. The host is the $I^2C$ master and the sensor hub is configured as the slave device. Since all communication on an $I^2C$ bus must be initiated by the master, an interrupt signal is added to the $I^2C$ bus, allowing the sensor hub to signal to the host that new data is available and to request the host to initiate a transfer of this data.

The 7-bit $I^2C$ slave address of the sensor hub is 0x18 (0b0011000). Please refer to UM10204 '$I^2C$-bus specification and user manual' for more information on $I^2C$.

Fig 19 shows the host interface from a hardware perspective.

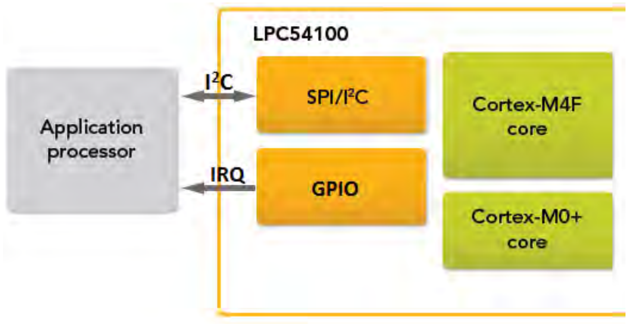


**Fig 19. Block diagram of the host interface hardware**

## 6.3 Protocol

The software protocol for the host interface has been designed in such a way that:

- The host can send commands to the sensor hub at any time by issuing a write on the $I^2C$ bus, as long as the previous command or data-read has finished before initiating a new command. Depending on which command has been issued by the host, the host may send additional parameters and may issue a read on the $I^2C$ bus to read the sensor hub's response.
- The host reads the data from the sensor hub after the sensor hub has asserted its IRQ signal. The read is performed in two steps. First, the host sends a 'read data length' command. After writing this command, the host issues a read on the $I^2C$ bus to get the number of bytes the sensor hub wants to transmit. Next, the host issues a 'get data' command, followed by a read of the exact number of bytes reported previously by the sensor hub to get the data from the sensor hub.

Fig 20 and Fig 21 show the signal timing diagrams for these two use-cases.

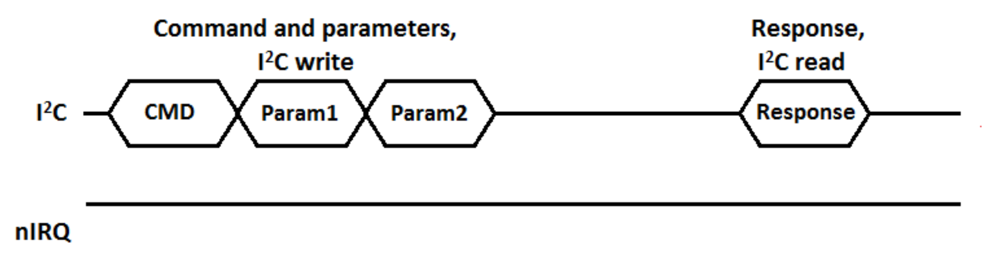


**Fig 20. Simplified signal timing diagram of host sending command and receiving response**

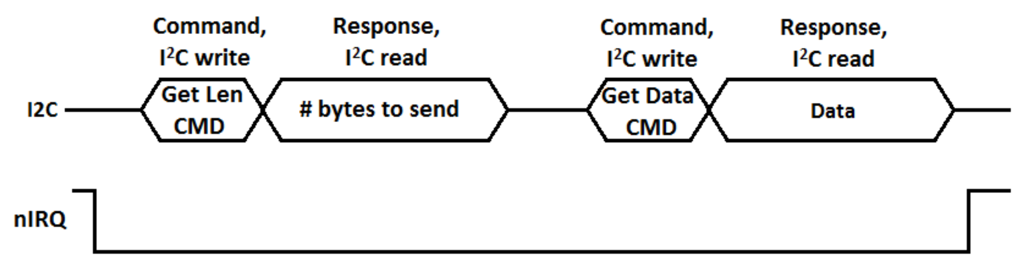


**Fig 21. Simplified signal timing diagram of sensor hub sending sensor data**

### 6.3.1 Commands, parameters, and responses

Commands are used to communicate from the host to the sensor, for example, to enable/disable sensor and to initiate a read of the sensor data. The command itself is an 8-bit opcode and can have up to 3 8-bit parameters. Some commands cause the sensor hub to send a response back to the host. A graphical representation of the commands and responses from the host to the sensor are shown in Fig 20 and Fig 21.

The following are the details of the commands that are implemented.

**Command: WHO_AM_I**

| | | |
|---|---|---|
| Description: | Sensor hub returns constant value (0x54) to be able to identify the sensor hub | |
| CMD: | 0x00 | |
| Parameters: | 0 parameters | |
| Response: | 1 byte: | |
| | 1st byte: | Constant value of 0x54 |

**Command: GET_VERSION**

Description: Get sensor hub firmware version

| | | |
|---|---|---|
| CMD: | 0x01 | |
| Parameters: | 0 parameters | |
| Response: | 2 bytes: | |
| | 1st byte: | Major version of sensor hub firmware |
| | 2nd byte: | Minor version of sensor hub firmware |

**Command: RESET**

| | |
|---|---|
| Description: | Disables all sensors |
| CMD: | 0x02 |
| Parameters: | 0 parameters |
| Response: | No response |

**Command: GET_DATA_LENGTH**

| | | |
|---|---|---|
| Description: | Send by host when sensor hub asserts its IRQ to get number of data bytes that the sensor hub wants to send to the host | |
| CMD: | 0x03 | |
| Parameters: | 0 parameters | |
| Response: | 2 bytes: | |
| | 1st byte: | Low byte of 16-bit data length (number of data bytes sensor hub wants to transmit) |
| | 2nd byte: | High byte of 16-bit data length (number of data bytes sensor hub wants to transmit) |

**Command: GET_DATA**

| | |
|---|---|
| Description: | Get data from sensor hub. Number of data bytes corresponds to the number reported in response to command 'GET_DATA_LENGTH' |
| CMD: | 0x04 |
| Parameters: | 0 parameters |
| Response: | *N* bytes as reported before in response to command 'GET_DATA_LENGTH' Refer to section '6.3.2. Sensor data' for more info on how sensor data is structured. |

**Command: SENSOR_ENABLE**

| | | |
|---|---|---|
| Description: | Enable/disable specified sensor | |
| CMD: | 0x20 | |
| Parameters: | 2 bytes: | |
| | 1st byte: | Virtual sensor ID of the sensor to enable/disable |
| | 2nd byte: | 0x00 to disable the specified sensor, != 0x00 to enable |
| Response: | No response | |

**Command: GET_SENSOR_STATE**

| | |
|---|---|
| Description: | Get enabled state (enabled/disabled) of specified sensor |
| CMD: | 0x21 |

Parameters: 1 byte:
- 1st byte: Virtual sensor ID of the sensor to get the enabled state of

Response: 1 byte:
- 1st byte: 0x00 if specified sensor is disabled, 0x01 if enabled

**Command: SET_DELAY**

Description: Configure sample interval (‘delay’) for the specified sensor

CMD: 0x22

Parameters: 3 bytes:
- 1st byte: Physical sensor ID of the sensor to configure the delay of
- 2nd byte: Low byte of 16-bit delay (in milliseconds)
- 3rd byte: High byte of 16-bit delay (in milliseconds)

Response: No response

**Command: GET_DELAY**

Description: Get sample interval (‘delay’) for the specified sensor

CMD: 0x23

Parameters: 1 byte:
- 1st byte: Physical sensor ID of the sensor to get the delay of

Response: 2 bytes:
- 1st byte: Low byte of 16-bit delay (in milliseconds)
- 2nd byte: High byte of 16-bit delay (in milliseconds)

### 6.3.2 Sensor data

New sensor samples are stored in the host interface transmit buffer until the host is ready to initiate the transfer, see section 4. SPM-S Sensor Data Acquisition. As a result, after the host issues a read, the transferred data is likely to contain multiple samples.

To be able to identify each sample, the actual sensor data is preceded by a header. This header identifies:

- Which sensor the sample comes from, using the sensor’s virtual sensor ID.
- When the sample was taken, using the sample’s timestamp.

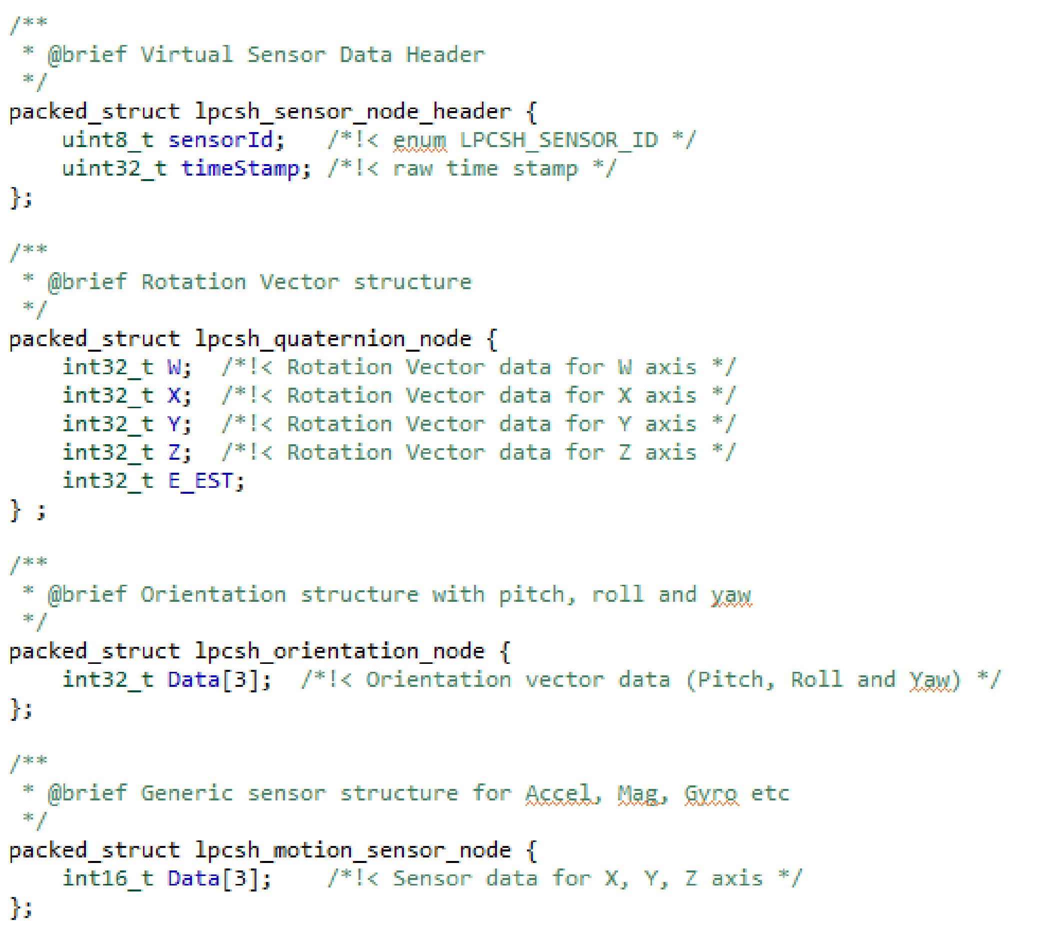
Fig 22 shows the format the host receives each sample.

| | | |
|---|---|---|
| 8-bit virtual sensor ID | 32-bit timestamp | Sensor data |

**Fig 22. All sensor data transmitted to the host is preceded by a 5-byte header**

The size and content of the actual sensor data depends on which sensor the data belongs to. Different sensors have different data structures and different structure sizes.

The structures for the generic header, the quaternion rotation vector, the orientation sensor and raw accelerometer/gyroscope/magnetometer data are shown in Fig 23. For structuring of the other sensors, please refer to the source code in file *hostif_protocol.h*.



```
/**
 * @brief Virtual Sensor Data Header
 */
packed_struct lpcsh_sensor_node_header {
    uint8_t sensorId;   /*!< enum LPCSH_SENSOR_ID */
    uint32_t timeStamp; /*!< raw time stamp */
};

/**
 * @brief Rotation Vector structure
 */
packed_struct lpcsh_quaternion_node {
    int32_t W;  /*!< Rotation Vector data for W axis */
    int32_t X;  /*!< Rotation Vector data for X axis */
    int32_t Y;  /*!< Rotation Vector data for Y axis */
    int32_t Z;  /*!< Rotation Vector data for Z axis */
    int32_t E_EST;
} ;

/**
 * @brief Orientation structure with pitch, roll and yaw
 */
packed_struct lpcsh_orientation_node {
    int32_t Data[3];  /*!< Orientation vector data (Pitch, Roll and Yaw) */
};

/**
 * @brief Generic sensor structure for Accel, Mag, Gyro etc
 */
packed_struct lpcsh_motion_sensor_node {
    int16_t Data[3];    /*!< Sensor data for X, Y, Z axis */
};
```

**Fig 23. Data structures for the generic header and several sensors**

## 6.4 Real-world example

To get a better understanding of the interaction between the sensor hub and the host, a schematic representation of what happens when the host enables a sensor is shown in Fig 24.

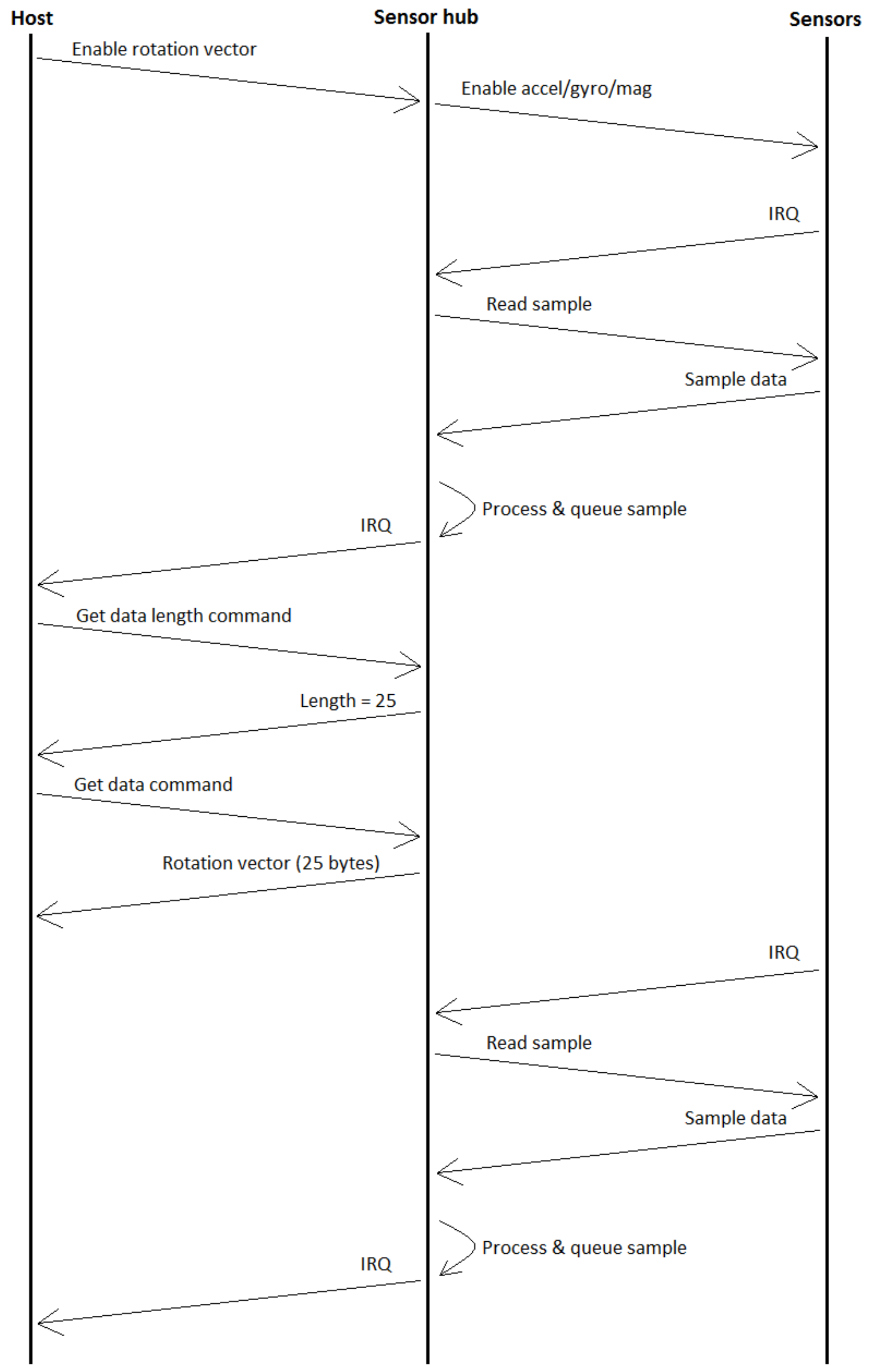


**Fig 24. Interaction between the host, sensor hub, and sensors**

# 7. Power optimization

Low power consumption is critical for a sensor hub solution because sensor hubs are typically used to achieve always-on sensor processing. Therefore, the NXP SPM-S is a power-optimized solution. This chapter discusses the common techniques to lower power consumption on embedded systems, explains how these techniques are put in practice in the SPM-S software, and provides power numbers measured on the SPM-S. This chapter will only focus on the MCU power. Power optimization of the full system is outside the scope of this application note.

## 7.1 Power-optimization techniques

The LPC54100 series is designed for low-power applications and offers many ways to optimize the software to reduce the power consumption to a minimum. This section explains several common techniques to lower the power consumption on microcontroller and how these techniques are applied to the LPC54102 sensor hub firmware.

Before continuing on the specific optimization techniques, it is important to know the difference between 'static power' and 'dynamic power'. Static power is the power consumed independent of the clock frequency, while dynamic power scales with the operating frequency. Together, they make up the full power consumption of the chip:

$$P_{total} = P_{static} + P_{dynamic}$$

$$P_{static} = I_{static} \times V$$

$$P_{dynamic} = A \times C \times V^2 \times F$$

In which:

- $I_{static}$ is the static current consumption.
- $V$ is the voltage used by the circuits on the chip (note this does not have to match the chip's external voltage supply, it may be converted down internally).
- $A$ is the activity factor, which is the probability of a gate experiencing an energy-consuming transition (low-to-high) in an arbitrary clock cycle.
- $C$ is the capacitance being charged/discharged.
- $F$ is the switching frequency.

From the above formulas it is clear that lowering the supply voltage results in a significant decrease in total power consumption. This section continues on how software can be optimized to reduce the power consumption.

### 7.1.1 Carefully choose which peripherals to enable when

The first step in power optimization is to enable the peripherals and blocks in the MCU that are actually going to be used. Microcontrollers often have ways to disable the clock to certain blocks ('clock gating') and disable the power to specific blocks ('power gating'). Clock gating will help reduce the dynamic power consumption while power gating will help reduce the static power consumption as well.

Most blocks/peripherals are not actively used all the time, but only periodically. Usually the power contribution of such peripherals is not significant, but in rare cases their power contribution is significant, and it may be beneficial to disable the peripheral when it is not being used. Often there is a time penalty associated with such schemes, for example, the time for a PLL to lock after re-enabling the PLL again.

The energy saved using such schemes can be calculated using the following formula:

$$E_{save} = P_{peripheral} \times t_{off} - P_{total} \times t_{penalty}$$

Where:

- $P_{peripheral}$ is the power consumption of the peripheral.
- $t_{off}$ is the time the peripheral will be off.
- $P_{total}$ is the total power consumption of the chip while enabling the peripheral.
- $t_{penalty}$ is the time it takes to enable the peripheral and any required delay.

In the SPM-S firmware, all peripherals that are not used throughout the runtime are clock-gated, and if possible, power gated at startup, thereby reducing both the static and dynamic power consumption.

Some code sections are executed at high CPU speed using the PLL as main clock source, while most code sections are executed at low speed, using the 12 MHz IRC as main clock source. With the formula mentioned in the above paragraph, one can calculate the power saved by turning the PLL off when it is not used:

$$E_{save} = P_{peripheral} \times t_{off} - P_{total} \times t_{penalty}$$

Assuming:

- $P_{peripheral} = 700\ \mu A \times 1.8V = 1.26\ mW$ (PLL current times VDD voltage).
- $t_{off} = 9\ ms$ (assuming 100 Hz 9-axis sensor fusion).
- $P_{total} = I \times V = 1.9\ mA \times 1.8V = 3.42\ mW$ (the total power consumption of the chip while enabling the peripheral [2]).
- $t_{penalty} = 75\ \mu sec$ (typical PLL lock time).

➔ $E_{save} = P_{peripheral} \times t_{off} - P_{total} \times t_{penalty} =$

$1.26\ mW \times 9\ ms - 3.42\ mW \times 75\ \mu sec = 11.34\ \mu J - 0.26\ \mu J = 11.08\ \mu J$

$E_{save}$ can be converted to the average saved power consumption:

$$P_{save,avg} = \frac{E_{save}}{T_{period}} = \frac{11.08\ \mu J}{10\ ms} = 1.11\ mW$$

### 7.1.2 Choosing the right low-power mode

Microcontrollers usually offer several low-power (sleep) modes. These modes allow power control in several steps, starting by shutting off the clock to the CPU, up to powering down the whole chip except for a small 'always-on' power domain. In any of these modes, the total power consumption of the chip is significantly lower than the active power consumption. Significant power can be saved by spending as much time as possible in any of the available low-power modes.

Deeper power modes have a lower power consumption, but usually at the cost of a higher wake-up time and less available wake-up sources. This means the properties of the low-power modes must be compared to the restrictions of the application (real-time handling of events, wake-up sources) and the wake-up time should be in the right

2. [2] Assuming VDD is 1.8V and IDD is 1.9 mA (taken from LPC5410x datasheet, supply current while executing CoreMark from flash at 12 MHz)

proportion of the saved power consumption. The time spent in the low-power mode is similar to the $E_{save}$ formula in section 7.1.1.
The LPC54102 defines four reduced power modes:

- Sleep mode. The clock to the core is shut off. Peripherals and memories are active and operational. Analog components can be turned on or off using use the PDRUNCFG register.
- Deep-sleep mode. The clock to all CPUs is shut down and the peripherals receive no internal clocks. All SRAM and registers maintain their internal states. The flash is in stand-by mode to minimize wake-up time, and the IRC is turned off to save power.
- Power-down mode. Same as deep-sleep mode, but the flash is now also powered down to conserve power at the expense of a somewhat longer wake-up time.
- Deep power-down mode. The entire system (CPUs and all peripherals) is shut down except for the PMU and the RTC. On wake-up, the part reboots.

Though deep-sleep, power-down, and deep power-down modes define which analog components are turned on or off, this can always be overwritten when entering these modes through the power profile API. This API allows the application code to specify which analog components must remain on while in the specified low-power mode.

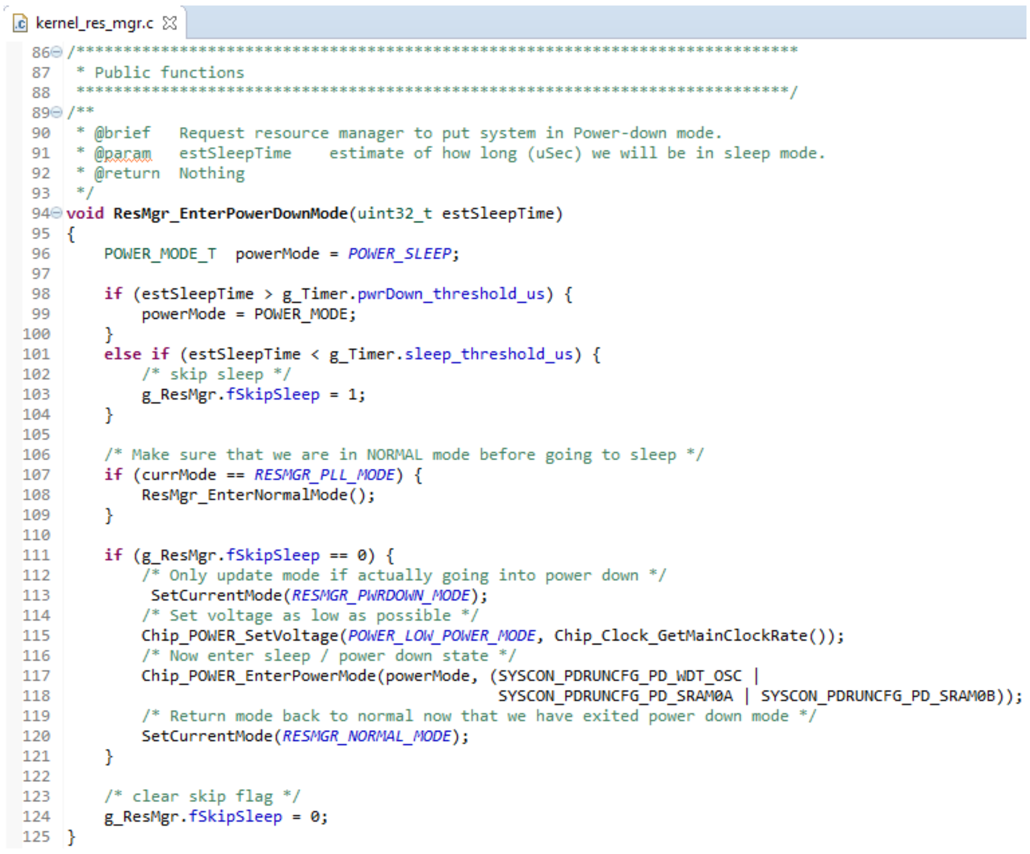
The power manager of the SPM-S framework has a single function to enter a low-power state, *ResMgr_EnterPowerDownMode()*. The function takes a single parameter, which is the estimated time to spend in the low-power mode. Depending on this parameter, the LPC54102 enters into either sleep mode or power-down mode:

- If the estimated sleep time is shorter than 400 µsec, the LPC54102 enters into regular sleep mode.
- If the estimated sleep time is longer than 400 µsec, the LPC54102 enters into power-down mode.

This assures the selection of the power mode that will result in the largest power saving. If the estimated sleep time is for a short period (< 400 µsec), the lower power consumption of the power-down mode will not weigh up against its longer wake-up time.

During power-down mode, the watchdog timer remains enabled as this low-power timer is used for time-stamping. Also SRAM0 is configured to be retained during power down.

Before entering the low power mode, another power API is used to configure the internal voltage regulator to its lowest setting.



kernel_res_mgr.c

```
/******************************************************************************
 * Public functions
 ******************************************************************************/
/**
 * @brief   Request resource manager to put system in Power-down mode.
 * @param   estSleepTime    estimate of how long (uSec) we will be in sleep mode.
 * @return  Nothing
 */
void ResMgr_EnterPowerDownMode(uint32_t estSleepTime)
{
    POWER_MODE_T  powerMode = POWER_SLEEP;

    if (estSleepTime > g_Timer.pwrDown_threshold_us) {
        powerMode = POWER_MODE;
    }
    else if (estSleepTime < g_Timer.sleep_threshold_us) {
        /* skip sleep */
        g_ResMgr.fSkipSleep = 1;
    }

    /* Make sure that we are in NORMAL mode before going to sleep */
    if (currMode == RESMGR_PLL_MODE) {
        ResMgr_EnterNormalMode();
    }

    if (g_ResMgr.fSkipSleep == 0) {
        /* Only update mode if actually going into power down */
         SetCurrentMode(RESMGR_PWRDOWN_MODE);
        /* Set voltage as low as possible */
        Chip_POWER_SetVoltage(POWER_LOW_POWER_MODE, Chip_Clock_GetMainClockRate());
        /* Now enter sleep / power down state */
        Chip_POWER_EnterPowerMode(powerMode, (SYSCON_PDRUNCFG_PD_WDT_OSC |
                                              SYSCON_PDRUNCFG_PD_SRAM0A | SYSCON_PDRUNCFG_PD_SRAM0B));
        /* Return mode back to normal now that we have exited power down mode */
        SetCurrentMode(RESMGR_NORMAL_MODE);
    }

    /* clear skip flag */
    g_ResMgr.fSkipSleep = 0;
}
```

**Fig 25. Function ResMgr_EnterPowerDownMode() takes care of entering the most lower power mode when the SPM-S firmware is idle**

### 7.1.3 Dynamic frequency scaling

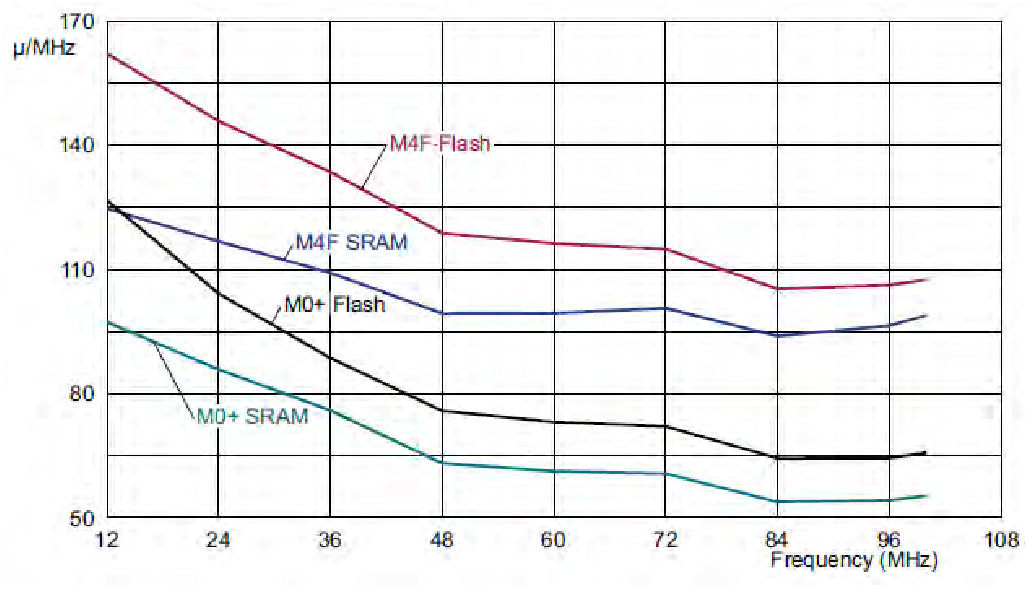
Besides choosing the right low-power mode, the software should also maximize the time spent in the low-power mode, by decreasing the time spend in active mode.

For certain tasks, such as algorithm execution, this can easily be done by increasing the CPU frequency. For these kind of tasks, instead of running at a fast-enough frequency to support the real-time needs, running at a higher frequency will:

- Reduce the time spent in active mode and increase the time spent in the low-power mode. The low-power mode significantly reduces both the static and dynamic power consumption compared to the power consumption in active mode, resulting in a lower average power consumption.
- In general, result in a lower current consumption per MHz (μA/MHz) while in active mode. By definition, the static power consumption is not (directly) impacted by the frequency. Since the dynamic power consumption scales linearly with the frequency and the static power consumption remains the same, the μA/MHz will decrease when increasing the frequency, as the constant static current consumption will get divided by a higher frequency. Note that though a higher frequency does not impact the

static current consumption directly, there usually is an increase in static and dynamic power, as the CPU often requires a higher voltage when running at higher frequencies. Also many MCUs rely on PLLs to increase the oscillator's frequency to a higher frequency, which consumes significant power when enabled. The µA/MHz profile of the LPC54102 is shown in Fig 26.

For such tasks it is beneficial to run the CPU at high core frequency.



Conditions: $V_{DD}$ = 3.3 V; $T_{amb}$ = 25 °C; active mode; all peripherals disabled; BOD disabled; Prefetch disabled in FLASHCFG register. System clock flash access time set by power API. SRAM0 powered, SRAM1 and SRAM2 powered down. Measured with Keil uVision 5.12. Optimization level 0, optimized for time off.

12 MHz: IRC enabled; PLL disabled. 24 MHz - 100 MHz: IRC enabled; PLL enabled.

**Fig 26. µA/MHz profile of the LPC54102 running CoreMark**

However, for other tasks, increasing the CPU frequency does not affect the execution time of that task because the execution relies on external, real-time events. An example of this would be the sensor hub's host interface task; the speed of the interface and the number of bytes to transfer dictates the execution time and increasing the CPU speed does not decrease the execution time. For such tasks it is best to choose the most low-power clock source (example, internal oscillator with no PLL) and run at the slowest speed possible that will still guarantee enough performance without delaying the task any further

By dynamically scaling the frequency for each task, significant power can be saved; time spent in low-power mode is maximized without unnecessary wasting of CPU cycles (thus power).

The SPM-S firmware has implemented dynamic frequency scaling:

- For all interrupts and ‘housekeeping’ tasks, the CPU is clocked from the 12 MHz IRC oscillator. These tasks are short by nature, or slowed down by external events (for example, sensor I$^2$C bus, or host interface bus). Running at a higher frequency does not significantly reduce the execution time, but would result in higher power consumption.
- For the sensor fusion algorithm, the CPU is clocked from the PLL, which has been configured to output a frequency of 84 MHz (sweet spot for LPC54102, see Fig 26). By executing this task at the most power efficient configuration (lowest µA/MHz), the task consumes the least amount of energy and the time spent in low-power mode is maximized.

Besides the dynamic frequency scaling, the internal voltage regulator of the LPC54102 is also dynamically scaled, minimizing the dynamic power and static current.

### 7.1.4 Software architecture

A solid (interrupt-driven) architecture can help achieve simple and effective power management:

- All interrupt handlers should be as short as possible, and only do the bare minimum required. For any significant processing, it is best to defer this to the main loop.
- For every iteration of the main loop, the MCU checks if work (deferred from the IRQ handlers) needs to be done. As soon as there are no tasks left in the main loop, the low-power mode is entered.
- Since no code in the interrupt handlers should benefit significantly from a faster CPU clock (since the IRQ handlers are short), all power management and clock switching can be done from the main loop.

The SPM-S framework has been designed on this architecture, resulting in simple and effective power management.

## 7.2 SPM-S power measurements

Fig 27 and Fig 28 show power plots of the LPC54102 running the SPM-S firmware. Conditions:

- LPCXpresso54102 with sensor board.
- SPM-S connected to PC using the debug probe USB connector.
- Teapot executable on the PC is acting as host for the sensor hub and active (i.e. reading data when *nIRQ* is asserted). The application has previously enabled the rotation vector sensor at 100 Hz rate.
- Power is measured using a Monsoon Power Monitor. Voltage is configured at 2.01V (lowest output voltage supported).
- Monsoon Power Monitor is connected to LPC54102 VDD pins, JP4.2 on the LPCXpresso54102 board (see Fig 29). Solder jumper JS6 is modified to be open.

Fig 27 shows the power over a period of 500ms. The peaks in the plot are caused by the execution of the sensor fusion algorithm at 84 MHz CPU clock.

Fig 28 shows the power over a shorter period of 40 ms. For 100 Hz sensor fusion, this means four iterations of sampling the sensors, processing the data, and transferring the data to the host, The power modes can be clearly distinguished in the plot:

- The peaks (~25 mW) are caused by the execution of the sensor fusion algorithm at 84 MHz CPU clock.
- The lower peaks (~5 mW) are mainly the moments when the sensors are being sampled and when the data is transferred to the host.
- The near-zero power is measured when the LPC54102 is in power-down mode.

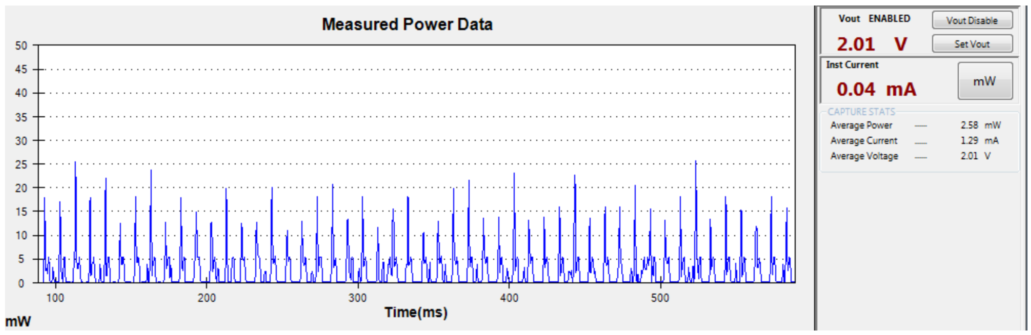


Fig 27. Power consumption of SPM-S running 100 Hz 9-axis sensor fusion. Average power consumption is 2.58 mW

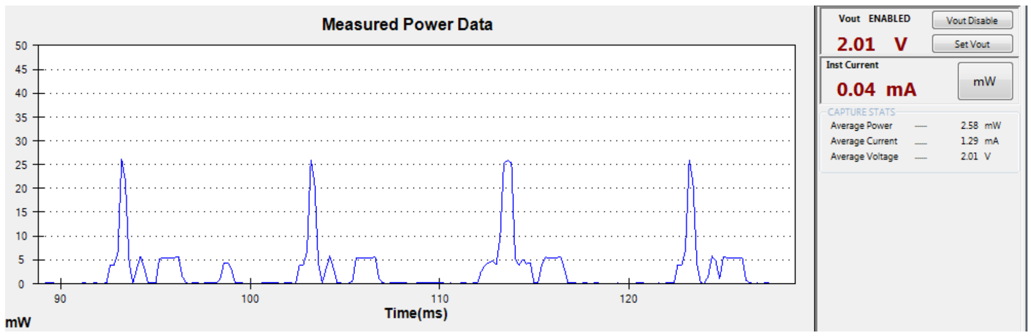


Fig 28. Detailed view of 4 iterations of the sensor sampling, sensor fusion and host interface transfer.

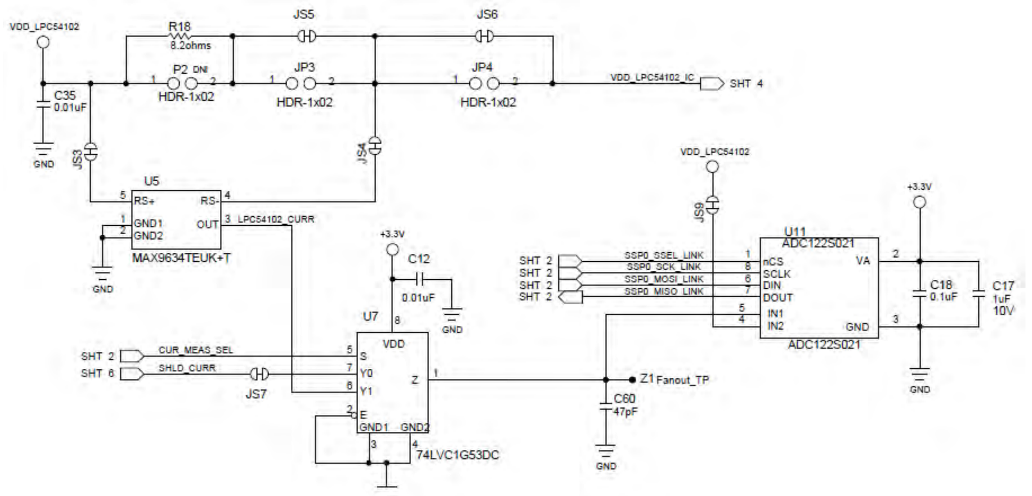


Fig 29. Snippet of LPCXpresso54102 board schematic. For the power measurements JS6 is opened and power is supplied using a Monsoon Power Monitor on JP4.2

## 8. Conclusion

The LPC54100 series are high performance, yet low-power dual-core ARM Cortex-M4/M0+ microcontrollers. Typical applications include low-power sensor hubs in mobile handsets and tablets, portable health and fitness monitoring devices, home and building automation products, robotics and flying drones/quadcopters.

The Sensor Processing/Motion Solution (SPM-S) showcases the LPC54102 in a typical sensor-hub application; the microcontroller can collect samples from various sensors, process the data using sensor fusion algorithms, and send the processed data to the host over the host interface.

Power-optimized source code for the microcontroller, a pre-compiled Bosch Sensortec BSXlite Sensor Fusion library, and a Windows based host-side application are freely available, allowing developers to quickly develop low power, always-on sensor-processing applications using the LPC54100 series.

# 9. Legal information

## 9.1 Definitions

**Draft —** The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

## 9.2 Disclaimers

**Limited warranty and liability —** Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

**Right to make changes —** NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use —** NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications —** Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Export control —** This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Translations —** A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Evaluation products —** This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

## 10. List of figures

## 11. List of tables

## 12. Contents