BHI260AB BHA260AB

Ultra-low power high performance Smart Sensor Hub with integrated sensors

BHI260AB-BHA260AB Programmer's Manual

Document revision	1.5
Document release date	May 28th, 2020
Document number	BST-BHI260AB-AN002-03
Technical reference code(s)	0 273 141 367 0 273 141 392
Notes	Data and descriptions in this document are subject to change without notice. Product photos and pictures are for illustration purposes only and may differ from the real product appearance.

Table of Contents

List of Figures

List of Tables

General description

This document describes the process of developing firmware for the BHI260AB and BHA260AB devices (hereafter referred to as the “BHy2”).

The BHy2 is a family of ultra-low-power smart hubs consisting of Bosch Sensortec’s new programmable 32-bit microcontroller (Fuser2), state-of-the-art motion sensors, and a powerful software framework with pre-installed sensor fusion and other sensor processing software in a small LGA package.

The firmware to run on the Fuser2 microcontroller is divided into a ROM image built into the BHy2 and RAM/Flash firmware images which can be used to customize firmware and provide patches for the ROM image.

The ROM firmware includes a bootloader, standard C, math, and security libraries, the host interface and support for basic host commands, and low -level hardware drivers.

When booting, the BHy2 bootloader loads a RAM/Flash image that defines the customization of the BHy2, and may contain additional algorithms defined by the user or Bosch Sensortec. It is possible to use the BHy2 device even without developing new firmware, since binary firmware files are provided on the Bosch Sensortec website, which provide broad functionality with both internal and external sensors, e.g. implementing a versatile ready-to-go Android™ sensor hub.

The purpose of this document is to describe how additional algorithms can be compiled into the firmware and how the firmware can be tailored to the application needs.

Figure 1 describes the internal architecture of the firmware stack.

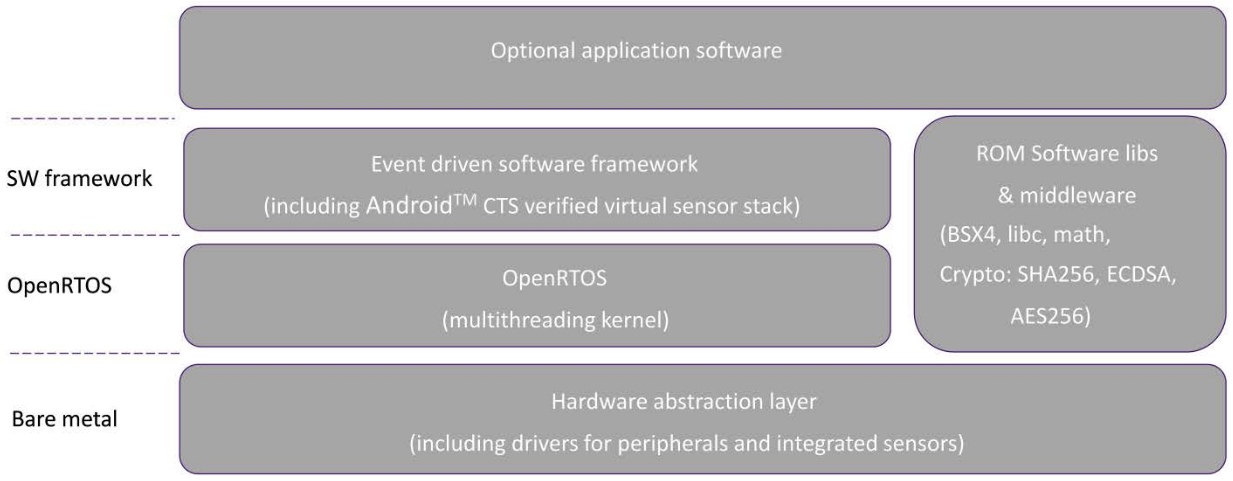

Optional application software
SW framework
Event driven software framework
(including Android™ CTS verified virtual sensor stack)
ROM Software libs
& middleware
(BSX4, libc, math,
Crypto: SHA256, ECDSA,
AES256)
OpenRTOS
OpenRTOS
(multithreading kernel)
Bare metal
Hardware abstraction layer
(including drivers for peripherals and integrated sensors)

Figure 1: Structure of the BHy2 Software Framework

In the default configuration, the firmware makes use of the event-driven software framework, and the customization can be done by adding or removing drivers to/from the event-driven system, or just configuring the drivers. This provides a high degree of flexibility, and most use cases can be handled with this level of configuration.

In addition to this document, Bosch Sensortec provides several documents which contain information on applications that are either directly or indirectly linked to firmware programming:

- BHI260AB/BHA260AB Datasheet: Provides technical details on the BHy2 devices.
- BHI260AB-BHA260AB SDK Quick Start Guide: Provides a simple and fast way to set up a firmware programming environment and get started with programming.
- BHI260AB-BHA260AB Evaluation Setup Guide: Contains information on how to evaluate BHy2 hardware using dedicated software on a host computer. This includes host channel communication with the BHy2 devices.
- BHI260AB-BHA260AB In-Circuit Debugging Guide: Provides methods for in-circuit debugging using the JTAG protocol in combination with a debugging environment.

1 Prerequisites and installation

The Software Development Kit (SDK) is designed for use on either a 64 bit Windows system or a 64-bit Linux system. Most testing has been done with Red Hat Enterprise Linux (7.4 or higher) and Ubuntu (16.04 LTS), however, no major issues are expected with other Windows or Linux distributions.

The following software packages are required for firmware development:

- BHy2 SDK release package from Bosch Sensortec
- Either of the ARC®[1] compiler toolchains
    - Synopsys MetaWare Compiler and Debugger Suite, revision 2018.03
    - ARC® GCC, revision 2018.09

    In general, the MetaWare toolchain is recommended due to increased testing and improved code size and execution speed.

Linux only: (utilities already included in binary format for Windows system)

- CMake revision 3.5 or higher
- Native C++ compiler and standard C library of the development system
- Ninja revision 1.3.4 or higher (optional)

Note that the toolchain versions provided in this document are those the SDK has been tested with. Newer versions of the toolchains can be installed but are not guaranteed to work as intended.

1.1 Compiler toolchains

The developer has a choice to use either the Synopsys MetaWare toolchain or the GNU GCC toolchain. MetaWare has shown to result in high code density and execution performance, so in general, it is the recommended choice. The GCC toolchain has the advantage of free availability without license costs incurred.

1.1.1 Obtaining and installing the Synopsys MetaWare C compiler

The latest version of the MetaWare Development Toolkit can be downloaded from the Synopsys website, see Reference 3, however, to obtain an older version of MetaWare, Synopsys must be contacted directly. To do so, please open a support case with Synopsys.

A valid license shall be obtained from Synopsys prior to installation.

1.1.1.1 Linux

The Synopsys MetaWare toolchain requires a 64-bit Linux operating system with 6 GB of free disk space to install.

Installation procedure:

1. Ensure the installer is executable.
```
chmod +w ./mw_devkit_arc_M_2018_03_linux_x64_install.bin
```
2. Run the installer.
```
./mw_devkit_arc_M_2018_03_linux_x64_install.bin -i console
```
3. Press ENTER to continue.
4. Read the EULA.
5. Accept the EULA.

[1] ARC® is a registered trademark of Synopsys.Inc.

6. Read the notice about multiple installations.
7. Select the installation path (ENTER for default).

   It is recommended to install it to the location */opt/synopsys/metaware/<version>/* if this is available to all users. For single-user installs, *~/metaware/<version>/* can be selected instead.
8. Verify installation options.
9. Perform the installation.
10. Read the licensing note and continue. Provide the license file or link to license server when requested.

    The installation is now complete.

Setting environment variables:

By default, the MetaWare installer sets the following variables by modifying the user start-up script:

```
METAWARE_ROOT: <install_path>/MetaWare
NSIM_HOME: <install_path>/nSIM/nSIM
LD_LIBRARY_PATH: <install_path>/Metaware/arc/bin
PATH: <install_path>/arc/bin;<install_path>/MetaWare/ide
```

These variables are set by modifying the user start-up script. They can be modified as needed. For a multi-user install, special care needs to be taken to ensure that all users have the appropriate environment variables set.

There are three primary methods for installing and activating the MetaWare license:

1. The license file can be installed to *<install_path>/license/arc.lic*.
2. The `SNPSLMD_LICENSE_FILE` environment variable can be set to point to the local license file.
3. The `SNPSLMD_LICENSE_FILE` environment variable can be setup to point to the license server by setting it to `<port>@<server>`.

1.1.1.2 Windows

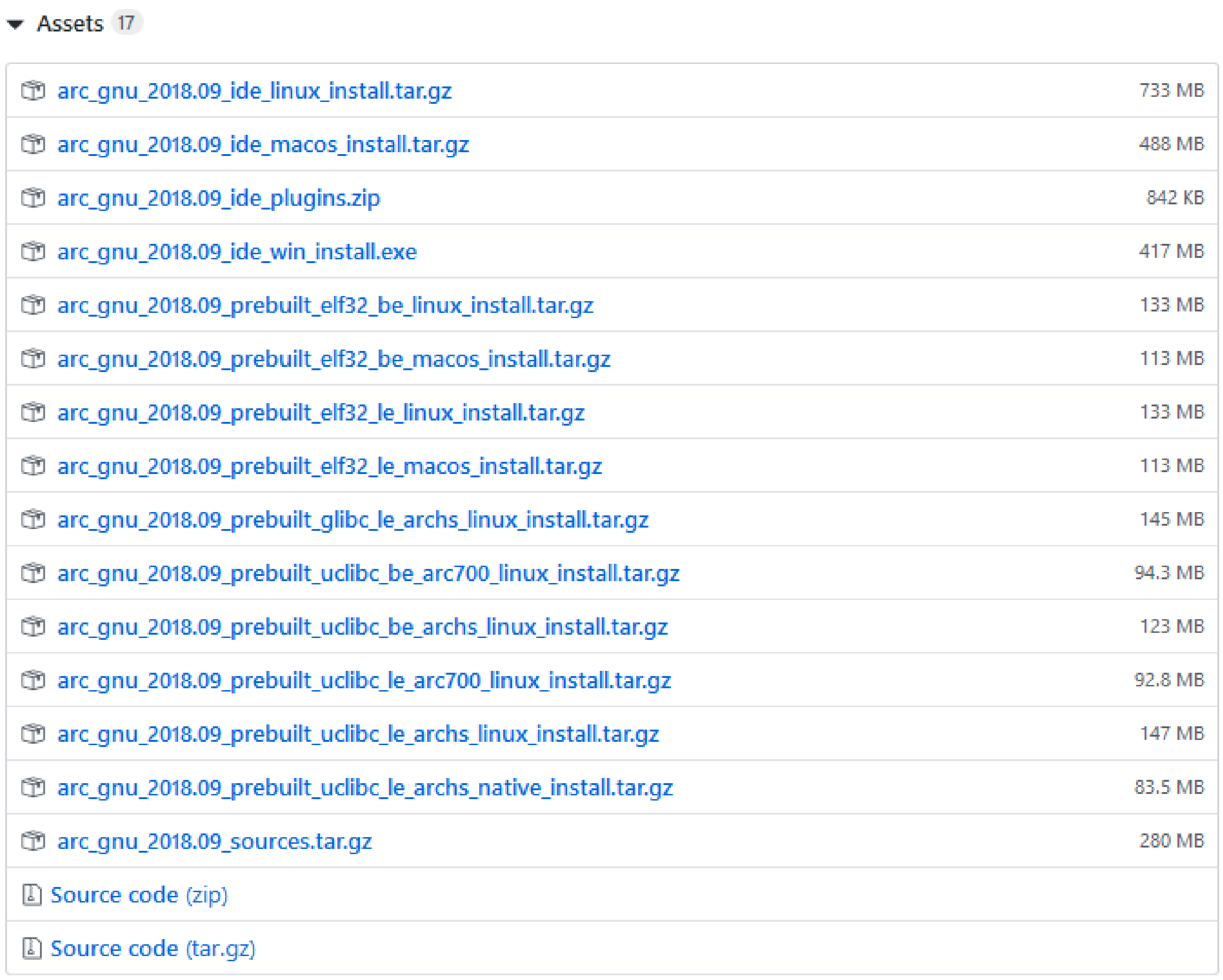
Installation procedure:

1. Run the installer by double-clicking the .exe file.
2. Read and accept the license agreement.
3. Select the install directory.
4. Verify installation options.
5. Perform the installation.
6. Read the licensing note and continue. Provide the license file or the link to the license server when requested.
7. Make sure that the environment variable SNPSLMD_LICENSE_FILE points to your MetaWare license file.

The installation is now complete.

1.1.2 Obtaining and installing the GNU C compiler for BHI260AB/BHA260AB

The latest version (as well as older versions) of the GNU Toolchain can be downloaded from the website given in Reference 4. A specific download package can be selected by using the drop down *Assets* in a specific release section, see Figure 2.



Assets 17

arc_gnu_2018.09_ide_linux_install.tar.gz	733 MB
arc_gnu_2018.09_ide_macos_install.tar.gz	488 MB
arc_gnu_2018.09_ide_plugins.zip	842 KB
arc_gnu_2018.09_ide_win_install.exe	417 MB
arc_gnu_2018.09_prebuilt_elf32_be_linux_install.tar.gz	133 MB
arc_gnu_2018.09_prebuilt_elf32_be_macos_install.tar.gz	113 MB
arc_gnu_2018.09_prebuilt_elf32_le_linux_install.tar.gz	133 MB
arc_gnu_2018.09_prebuilt_elf32_le_macos_install.tar.gz	113 MB
arc_gnu_2018.09_prebuilt_glibc_le_archs_linux_install.tar.gz	145 MB
arc_gnu_2018.09_prebuilt_uclibc_be_arc700_linux_install.tar.gz	94.3 MB
arc_gnu_2018.09_prebuilt_uclibc_be_archs_linux_install.tar.gz	123 MB
arc_gnu_2018.09_prebuilt_uclibc_le_arc700_linux_install.tar.gz	92.8 MB
arc_gnu_2018.09_prebuilt_uclibc_le_archs_linux_install.tar.gz	147 MB
arc_gnu_2018.09_prebuilt_uclibc_le_archs_native_install.tar.gz	83.5 MB
arc_gnu_2018.09_sources.tar.gz	280 MB
Source code (zip)	
Source code (tar.gz)	

Figure 2: ARC GNU Toolchain Download

1.1.2.1 Linux

The GNU toolchain requires a 64-bit Linux operating system. . The package with support for elf32 little-endian hosts is required. For the 2018.09 release, the correct version to install is the `arc_gnu_2018.09_prebuilt_elf32_le_linux_install.tar.gz` package.

Note:

The `arc_gnu_2018.09_ide_linux_install.tar.gz` file can be used instead if the eclipse ide is desired.

1. Extract the installation package.

   ```
   tar -xvf arc_gnu_2018.09_prebuilt_elf32_le_linux_install.tar.gz
   ```

2. Move the extracted folder to the installation path. For multi-user installs */opt/arc_gnu/<version>* is a good path, while single-user installs can use *~/arc_gnu/<version>*.

   ```
   mkdir -p ~/arc_gnu/
   mv arc_gnu_2018.09_prebuilt_elf32_le_linux_install ~/arc_gnu/2018.09
   ```

3. Update the PATH variable to include the *<install_path>/bin/* directory. This can be done by modifying the shell start-up script as appropriate.

1.1.2.2 Windows

The latest version of the GNU Toolchain can be downloaded from the website given in Reference 4. For the 2018.09 release, the correct version to install is the `arc_gnu_2018.09_ide_win_install.exe` package (see Figure 2).

Simply double-click the installer and follow the instructions in order to install the compiler toolchain.

1.2 Software Development Kit for BHI260AB/BHA260AB

1.2.1 Linux

The SDK is a shell archive that can be extracted under Linux. To install the SDK, execute the following steps:

1. Obtain the SDK shell archive for Linux.
2. Make it executable.

   ```
   chmod a+x BHI260_SDK_V1.0.0_Install.sh
   ```
3. Execute the shell script.

   ```
   ./BHI260_SDK_V1.0.0_Install.sh
   ```
4. Read and accept the license.
5. Select the installation directory.

Now the SDK tree is extracted to the selected directory.

1.2.2 Windows

To install the SDK, execute the following steps:

1. Obtain the SDK executable file for Windows.
2. Double-click BHI260_SDK_Installer_V1.0.0.exe.
3. Read and accept the license agreement, click *Next*.
4. Choose the install directory.

Now the SDK tree is extracted to the selected directory.

2 SDK structure and features

2.1 SDK and firmware structure

2.1.1 Overview of SDK structure

The SDK package contains all the necessary files for custom code development and linking. Typical custom code development will utilize the *drivers_custom* subdirectory (see Table 1 below).

Table 1: SDK Directory Structure

SDK File/Directory	Notes
apps	Directory which contains source code for applications that run outside of the sensor framework
boards	Configuration files for supported development boards and sensors
cmake	CMake files used to build the SDK
common	Source code for initialization code and reference header files, main CMake global configuration file
docs	Information regarding application notes
drivers	Source code and binary code of sensor drivers, custom drivers can be placed in this directory
drivers_custom	Driver templates for custom drivers. Custom driver code should be placed in this directory
gdb	Support files for using gdb
kernel	Binaries and objects files of the firmware kernel image
libs	Linkable binary image and header files for API libraries
user	Entry code for user-mode firmware, source code for custom user-mode RAM patches
utils	Executable image manipulation utilities, command line interface
win64	Binaries of utilities for Windows system.
build.sh	Shell script used to set up the build directory and build the firmware on Linux
build.bat	Batch script used to set up build directory and build the firmware on Windows
README.txt	Hints on requirements and instructions for building firmware
release	Created during build process, will contain final binary firmware files and elf files
build	Created during build process, will contain intermediate build artifacts, used memory resources for different board configurations etc.

2.1.2 Overview of firmware structure

Firmware images for the BHy2 are divided into a ROM image built into the BHy2 chip and RAM/Flash firmware images which can be used to customize firmware and provide patches for the ROM firmware. The ROM firmware includes the bootloader, SHA256 and ECDSA security libraries, the host interface and support for basic host commands, the BSX sensor fusion library and low-level hardware drivers.

RAM/Flash firmware images are further divided into two images – a kernel image and a user image, which are described below.

2.1.2.1 Kernel Mode and User Mode RAM Images

RAM/Flash firmware images are divided into two images - kernel image and user image. The kernel image includes the sensor framework, RTOS libraries, the sensor fusion library, support for additional host commands, and any necessary ROM patches. The user image includes all physical and virtual drivers. Table 2 describes various firmware components and their operating modes.

Kernel and user operating modes are used to provide privileged and restricted access to certain privileged instructions, registers, and memory locations. Kernel mode has the highest level of privilege and allows unrestricted access to privileged resources, while user mode has the lowest level of privilege and allows only restricted access to privileged resources. Any attempt to access privileged resources will result in a privilege violation exception.

Table 2: Firmware Components

Component	Image	Operating Context	Operating mode
Hooks	Kernel or user payload	Any	Kernel
Commands	Kernel payload	Interrupt/Host task	Kernel
Param IO	Kernel payload	Interrupt	Kernel
Physical Drivers	User payload	Interrupt/Sensor Framework task	Kernel
Virtual Drivers	User payload	Virtual Tasks	User

2.1.3 Available memory resources for custom Code

Figure 3 shows the breakdown of memory resources available for custom programing using program and data RAM. There is a total of 256 KB available RAM. One bank of 16 KB is dedicated program RAM and one bank of 16 KB is dedicated data RAM. The remaining 7 – 32 KB banks of RAM can be divided as needed between program and data RAM. During initialization, the firmware determines how many banks of program and data RAM are needed and allocates and powers on only what is needed.

In order to optimize for both size and speed, custom code should utilize available API functions. Available API functions abstract away access to hardware and are optimized for speed.

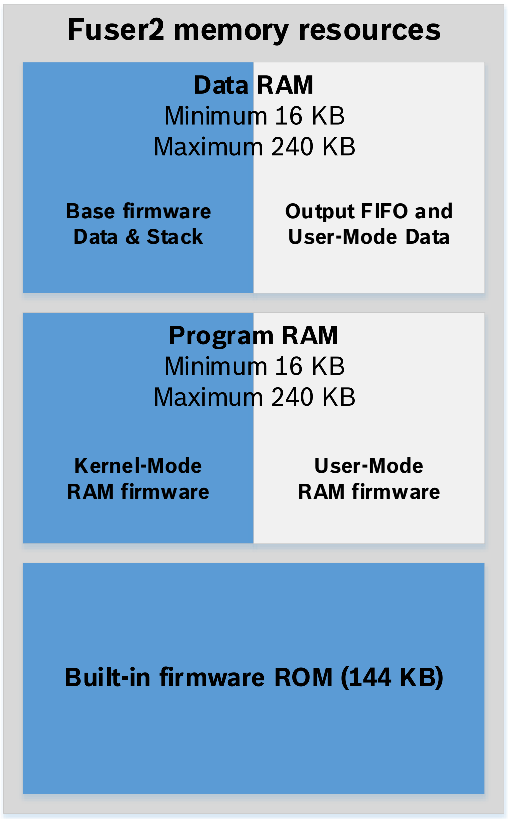


Figure 3: Available Memory Resources for Custom Code Development (White) and Reserved Memory (Blue)

2.2 Firmware configuration (using board configuration file)

Board configuration files are used to specify configuration data for different firmware builds. A board configuration file includes a global configuration section, a physical driver configuration section and a virtual driver configuration section. Comments begin with a hash mark (#) and go to the end of the current line.

Board configuration files can be found in the directory *boards* in the SDK root. The SDK contains multiple board configuration files. The main CMake file, *common/config.7189_di03_rtos_bhi260.cmake,* controls which board configurations are compiled during the firmware build process (see section 2.3.2).

2.2.1 Global configuration

The global configuration section specifies a number of hardware configurations. Each line in the global configuration section begins with the option name and is followed by the value(s) to be assigned to that option. The option name and value(s) are listed as comma separated values.

Table 3: Board Configuration – Global Values

Option	Value
stuffelf	stuffelf version used to write configuration file
irq	Host IRQ pin number
evcfg	12 values specifying the GPIO event configuration (events 1-11). Each value specifies whether the respective GPIO event interrupt gets its source per configuration 1 (0) or configuration 2 (1). The first value (event 0) is reserved and should be set to 0.
pull	Up to 28 values specifying the GPIO pull configuration, off = pull is disabled, on = pull is enabled. If no values are indicated, the default value will be off.
gpio	Up to 25 values specifying the GPIO value configuration, low = active low, high = active high, hiz = high impedance. If no values are indicated, the default value will be hiz.
sif_cfg	Sensor Interface selection, selects the protocol for SIF0 and SIF1. SIF2 is always connected to I2C master 1.

Option	Value
	sif_cfg: SIF0 SIF1 SIF2 0: SPI master 0 SPI master 1 I2C master 1 1: SPI master 0 I2C master 0 I2C master 1 2: I2C master 0 SPI master 1 I2C master 1
hif_disable	Host interface configuration. 0 = enabled, 1 = disabled
rom_name	Custom ROM firmware image name
hw	Target hardware name
fifo	FIFO allocation percentage, wake-up FIFO is allocated 50% plus half of this value, non-wake-up FIFO is allocated remaining. Valid values range from -100 (100% non-wake-up) to +100 (100% wake-up).
wordsreq	Number of words required by the FIFO
turbo	Run in turbo mode
rom	Expected ROM version (unused)
version	Custom user firmware version
build_type	Firmware build type, can be set to RAM, Flash, or both
config_list	BSX specification list file
config_spec	BSX specification file
ram_patches (optional)	Names of ram patches (hooks) that shall be included for this board
lib (optional)	Names of libraries that shall be included for this board

2.2.2 Physical drivers

The physical driver section includes one line for each physical driver that should be included in the firmware image. The configuration for each physical driver is specified in a comma separated list (one line per included driver) and includes the following values. No option names are included in the physical driver configuration list.

Table 4: Board Configuration – Physical Driver values

Option	Value
Driver ID	Driver ID of the physical driver to be included. It must match the driver ID indicated in the CMakeLists.txt file.
Sensor Bus Interface	The value must be one of none, i2c0, i2c1, spi0, or spi1.
Sensor Bus Address	If I2C, this is a 7-bit slave address (MSB = 0). If SPI, this is the GPIO pin for chip select.
GPIO Pin	GPIO IRQ pin number, specify a dash (-) if a GPIO IRQ pin does not exist
Calibration Matrix	9 floating point values indicating the calibration matrix used for the sensor
Calibration Offset	3 floating point values indicating the calibration offset for the sensor
Max rate	Override the maximum rate for the physical driver
Range	Set the dynamic range for the physical driver

For example, the BHA260AB Accel driver might be configured with this line in the physical driver configuration list.

```
45,spi0,25,2, 1, 0, 0, 0,-1, 0, 0, 0,-1, 0, 0, 0, -1.000000, 0
```

Figure 4: Example Physical Driver Configuration

2.2.3 Virtual drivers

The virtual driver section includes one line for each virtual driver that should be included in the firmware image. The configuration for each virtual driver is specified in a comma separated list (one line per included driver) and includes the following values. No option names are included in the virtual driver configuration list.

Table 5: Board Configuration – Virtual Driver values

Option	Value
Driver ID	Driver ID of the virtual driver to be included. It must match the driver ID indicated in the CMakeLists.txt file.
Max rate	Override the maximum rate for the virtual driver

For example, the virtual accel raw driver might be configured with this line in the virtual driver configuration list.

```
203, 400.000000 # accel raw depends on a virtual BSX source.
```

Figure 5: Example Virtual Driver Configuration

2.3 Build system and build targets

2.3.1 Compiling firmware

This section describes the firmware compilation process using the SDK. This compilation process is common for custom hooks and custom driver development. Note that compilation is a multi-step process – first, the source code is compiled and linked into *$SDK/user/<board>.elf* file. Second, the proper rotation matrixes and pin settings are applied using `stuffelf` utility. Finally the *.elf file is converted to the binary firmware format. The final firmware image is named `<board>.fw` and can be found in the *$SDK/release/fw* directory (*$SDK/release/gccfw* for GCC compilation*)*.

2.3.1.1 Setting environment for compilation

In order to identify the version of firmware running on the device, the `USER_VERSION` field in the Config Data Structure is populated during compilation. This value is readable from the User Version register during normal operation (see section 13 of BHI260AB/BHA260AB Datasheet, Reference 1 and Reference 2). The `USER_VERSION` field is set as shown in Table 6.

Table 6: RAM Version Setting in SDK Compilation Result

SDK checked from git	RAM Version Setting
Yes	Number of git commits in the current git repository (git rev-list HEAD)
No	Value from $SDK/config.cmake REVISION

2.3.1.2 Firmware generation for supported boards

A simplified compilation flow is available for standard boards. The build system utilizes CMake and can generate Ninja build files (default) or standard UNIX makefiles (if Ninja is not installed). There are build scripts (build.sh (Linux) and build.bat

(Windows)) in the $SDK directory that automates the initial build. To build standard firmware for all pre-defined boards without custom hooks or drivers, execute the following steps.

Linux:

```
cd $SDK
./build.sh
```

Windows:

```
cd $SDK
build.bat
```

The resulting binary firmware files are located in the *$SDK/build/user* directory. As a final step of the build process, a condensed version of the final artifacts is copied to the *$SDK/release* directory. The generated firmware images are located in *$SDK/release/fw* (*$SDK/release/gccfw* for GCC compilation*)*.

Individual board firmware can be built by specifiying a `<board-name>` as an argument of the build script, where `<board-name>` is the name of a board config file minus the '.cfg' extension.

Linux:

```
cd $SDK
./build.sh <board-name>
```

Windows:

```
cd $SDK
build.bat <board-name>
```

To create a new board file, copy an existing reference board file in the `$SDK/boards` directory to a new board file. Edit *$SDK/common/config.<$SDK_TYPE>.cmake* to add the name of the board file to the `BOARDS` variable. Be sure to choose a reference board file according to the sensors you need, editing it to use the correct GPIO pin and sensor drivers, and other configuration.

2.3.2 Configuring the firmware build (using the main CMake file)

The firmware build process can be customized by modifying the main CMake file, which is located in *$SDK/common/config.<$SDK_TYPE>.cmake*. Table 7 shows the main configurable parameters that need to be modified when developing custom firmware. In the `set` call, each added value must be written to a new line.

Table 7: Main CMake File Parameters

Parameter	Note
`BOARDS`	Contain the names of boards for which a firmware file shall be built. The name must match the board config file minus the '.cfg' extension.
`DRIVERS_NO_SOURCE`	Contain drivers that are not present as source code, which shall be included in the firmware. The name must match the driver folder and *.c name.
`ENABLED_DRIVERS`	Contain drivers that are present as source code, which shall be included in the firmware. The name must match the matching driver folder and *.c name.
`RAM_PATCHES`	Need to contain the names of the *.c files (located in *$SDK/user/RamPatches*) that contain hooks which shall be implemented.

2.3.3 Selecting the toolchain

The firmware build process automatically searches for installed compiler toolchains (GCC and/or MetaWare). In case both are found, MetaWare is chosen as the compiler.

If the build process shall use GCC in any cases, the build script can be called as follows:

Linux:

```
cd $SDK
./build.sh USE_GCC
```

Windows:

```
cd $SDK

build.bat USE_GCC
```

3 BHy2 driver architecture

This chapter provides information about how sensor data is handled on the BHy2 device in general. Furthermore, it explains how additional sensors and custom algorithms working on physical or virtual data can be integrated into the sensor framework.

In general, all functionality related to sensors and sensor data is handled by the sensor framework. This framework handles priorities among implemented drivers and organizes the general flow of data. The sensor framework is the entry point to the BHy2 custom software development.

The BSX sensor fusion library gets physical sensor data as input and provides calibrated, combined and raw data, which is either used as direct output data or as a data source for different algorithms, including custom algorithms. Algorithms that use BSX output as an input can be implemented in virtual drivers.

In general, the BSX sensor fusion library provides three output gates that are either used by algorithms or for sending sensor data directly to FIFO: Wake-up, Non-Wake-up and custom output gates.

3.1 General flow of data

Figure 6 provides an overview of the structure and dataflow of the BHy2 device.

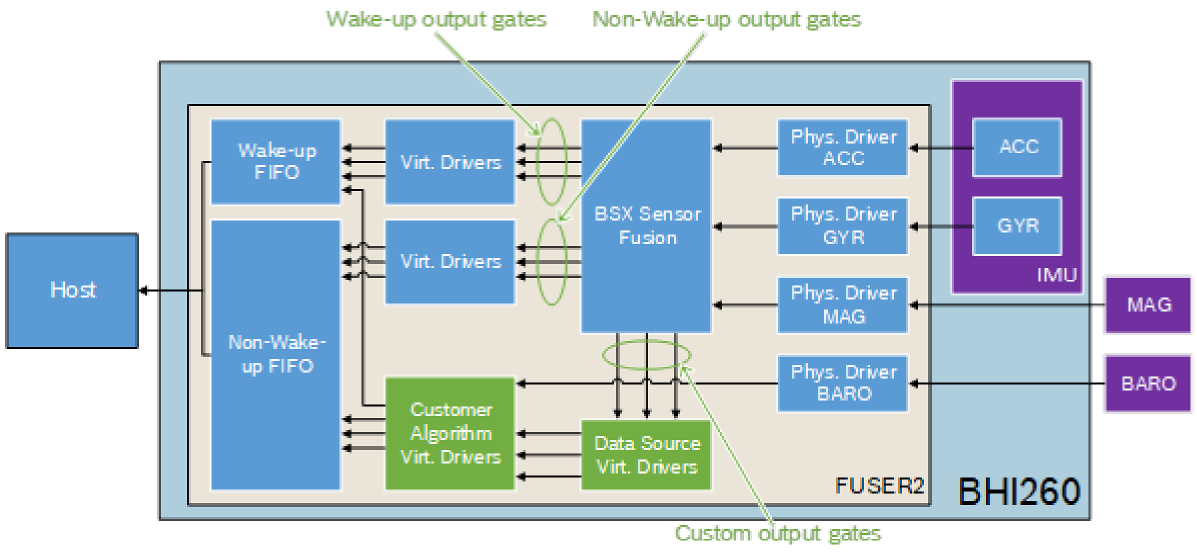


Figure 6: BHy2 Driver Architecture

Physical sensor data coming from internal or external sensors is accessed by one physical driver for each physical sensor. For a predefined set of sensors (accelerometer, magnetometer and gyroscope), the BSX fusion library uses the output of the physical driver and fuses this data. The BSX provides the processed data to virtual drivers, which are then used to compute additional algorithms or directly transfer the data to the FIFO. The sensor framework handles the transfer of data to either the Non-Wake-up FIFO or the Wake-up FIFO, which is accessible from the host side. For more information on the FIFO concept, see Reference 1 and Reference 2.

The BSX fusion library can also output raw data from the physical drivers to custom output gates, which can then be manipulated by custom algorithms. In order to access this data, a data source driver, which provides an interface to the BSX output data via custom output gates, has to be created. This data source driver can then be used as an input for custom algorithms implemented in virtual drivers.

Physical sensor data can also be accessed without using the BSX fusion library, by creating a pair of custom physical and virtual drivers. The output of these virtual sensors are then sent to the FIFOs, which is handled by the sensor framework.

Note that only physical sensors, which are not handled by the BSX, can be used for this concept. It is possible to attach multiple virtual drivers to one physical driver.

A virtual sensor must report sensor data to the host using the `reportSensorEvent` function. This function sends the output data to the host interface. If the sensor is explicitly enabled by the host, the output data will be inserted into the proper FIFO (non-wakeup FIFO and/or wakeup FIFO). If the sensor framework (via the trigger list/dependency chain) determined that a virtual sensor needs to be enabled, then the host interface will instead discard the output data.

4 Software development for BHI260AB/BHA260AB using the software framework

BHy2 functionality can be enhanced by writing custom sensor drivers that can be tied with particular hardware connected to the BHy2 as physical sensors or based on software as virtual sensors. Only virtual sensors can output data to the FIFO for transfer to the host, so users creating a non-standard physical sensor driver will need to also create a matching virtual sensor driver if they intend the data to be sent to the host. This section describes all steps necessary to write custom drivers.

4.1 Sensor driver overview

The BHy2 supports several sensor driver types that can depend on each other, be triggered by various sources and executed in different priority levels.

4.1.1 Sensor driver types

There are three basic types of sensor driver: Physical, Virtual and Timer. Note that any virtual driver can have an equivalent physical driver.

4.1.1.1 Physical sensors

- Providers of sensor data
- Triggered by GPIO interrupts
- Can be periodically polled using a timer if no GPIO is available
- Do not output data to the host

4.1.1.2 Virtual sensors

- Consumers of sensor data
- Can be triggered by any sensor driver type (physical, virtual, or timer)
- Can also be programmatically triggered to fork a new thread
- Special case of a virtual sensor is a programmatically triggered sensor

4.1.1.3 Timer sensors

- Providers or consumers of sensor data
- Timer-based virtual sensors
- Triggered by internal timer routines, automatically scheduled
- Special case of a timer sensor is a continuously triggered sensor (0Hz timer rate); this sensor cannot have any children and should be Priority M to work properly
- Limited to run below 64 KHz
- Allowed to output data to the host with the limitation presented in section 4.1.2

4.1.2 Predefined sensors

The BHy2 offers several predefined physical and virtual sensor types in addition to the possibility of adding new driver types.

4.1.2.1 Physical sensor types

Supported predefined physical sensor types are defined in

$SDK/libs/BSX/includes/bsx_physical_sensor_identifier.h and *$SDK/libs/SensorInterface/includes/SensorAPI.h* and include the following types:

- `BSX_INPUT_ID_ACCELERATION                  (1)`
- `BSX_INPUT_ID_ANGULARRATE                   (3)`
- `BSX_INPUT_ID_MAGNETICFIELD                 (5)`
- `BSX_INPUT_ID_TEMPERATURE_GYROSCOPE         (7)`

BSX_INPUT_ID_ANYMOTION	(9)
BSX_INPUT_ID_PRESSURE	(11)
BSX_INPUT_ID_POSITION	(13)
BSX_INPUT_ID_HUMIDITY	(15)
BSX_INPUT_ID_TEMPERATURE	(17)
BSX_INPUT_ID_GASRESISTOR	(19)
SENSOR_TYPE_INPUT_STEP_COUNTER	(0x20)
SENSOR_TYPE_INPUT_STEP_DETECTOR	(0x21)
SENSOR_TYPE_INPUT_SIGNIFICANT_MOTION	(0x22)
SENSOR_TYPE_INPUT_ANY_MOTION	(0x23)
SENSOR_TYPE_INPUT_EXCAMERA	(0x24)
SENSOR_TYPE_INPUT_GPS	(0x30)
SENSOR_TYPE_INPUT_LIGHT	(0x31)
SENSOR_TYPE_INPUT_PROXIMITY	(0x32)

4.1.2.2 *Virtual sensor types*

Supported predefined virtual sensor types are defined in *$SDK/libs/SensorInterface/includes/SensorAPI.h* and *$SDK/libs/BSX/includes/bsx_virtual_sensor_identifier.h*. The virtual sensor types in *bsx_virtual_sensor_identifier.h* are reserved and not available for custom use when defining new virtual sensors.
Key virtual sensor types are listed below. For an all-inclusive list, see the SDK source files mentioned above.

SENSOR_TYPE_TEMPERATURE	(0x80)
SENSOR_TYPE_PRESSURE	(0x81)
SENSOR_TYPE_HUMIDITY	(0x82)
SENSOR_TYPE_GAS	(0x83)
SENSOR_TYPE_WAKE_TEMPERATURE	(0x84)
SENSOR_TYPE_WAKE_PRESSURE	(0x85)
SENSOR_TYPE_WAKE_HUMIDITY	(0x86)
SENSOR_TYPE_WAKE_GAS	(0x87)
SENSOR_TYPE_STEP_COUNTER	(0x88)
SENSOR_TYPE_STEP_DETECTOR	(0x89)
SENSOR_TYPE_SIGNIFICANT_MOTION	(0x8A)
SENSOR_TYPE_WAKE_STEP_COUNTER	(0x8B)
SENSOR_TYPE_WAKE_STEP_DETECTOR	(0x8C)
SENSOR_TYPE_WAKE_SIGNIFICANT_MOTION	(0x8D)
SENSOR_TYPE_ANY_MOTION	(0x8E)
SENSOR_TYPE_WAKE_ANY_MOTION	(0x8F)
SENSOR_TYPE_VIRT_EXCAMERA	(0x90)
SENSOR_TYPE_GPS	(0x91)
BSX_OUTPUT_ID_ACCELERATION_PASSTHROUGH	(2)
BSX_OUTPUT_ID_ACCELERATION_RAW	(6)
BSX_OUTPUT_ID_ACCELERATION_CORRECTED	(8)
BSX_OUTPUT_ID_ANGULARRATE_PASSTHROUGH	(20)
BSX_OUTPUT_ID_ANGULARRATE_RAW	(24)
BSX_OUTPUT_ID_ANGULARRATE_CORRECTED	(26)
BSX_OUTPUT_ID_MAGNETICFIELD_PASSTHROUGH	(38)
BSX_OUTPUT_ID_MAGNETICFIELD_RAW	(42)
BSX_OUTPUT_ID_MAGNETICFIELD_CORRECTED	(44)
BSX_OUTPUT_ID_GRAVITY	(56)

- `BSX_OUTPUT_ID_LINEARACCELERATION` (62)

4.1.2.3 User provided physical sensor types

Custom physical sensor types may be defined in *$SDK/libs/SensorInterface/includes/SensorAPI.h*.

Custom sensor type values must be defined in the range of `NON_BSX_INPUT_ID_BEGIN` and `NON_BSX_INPUT_ID_END`. Do not use any already allocated value.

- `NON_BSX_INPUT_ID_BEGIN` (0x20)
- …
- `NON_BSX_INPUT_ID_END` (0x3F)

4.1.2.4 User provided virtual sensor types

Custom virtual sensor types may be defined in *$SDK/libs/SensorInterface/includes/SensorAPI.h*. Custom sensor type values must be defined between the values of `SENSOR_TYPE_CUSTOMER_VISIBLE_START` and `SENSOR_TYPE_CUSTOMER_VISIBLE_END`.

- `SENSOR_TYPE_CUSTOMER_VISIBLE_START` (0xA0)
- …
- `SENSOR_TYPE_CUSTOMER_VISIBLE_END` (0xBF)

4.1.3 Sensor priority level

One of the BHy2 features is support for multiple sensor priority levels. This guarantees that execution of the `handle_sensor_data` functions of triggered sensors is not mixed together – each priority level is executed at a different time. This feature brings greater flexibility in creating consecutive sensor series.

The base priority levels supported by the BHy2 are:

- PRIORITY_1
  - o Default value for physical sensors
  - o Executed as soon as GPIO interrupt occurs
- PRIORITY_2 - PRIORITY_4
  - o Used by virtual and timer sensors
  - o Executed when handling sensor data
- PRIORITY_M
  - o Lowest priority level
  - o Used by virtual and timer sensors
  - o Executed in the main execution routine

4.1.4 Sensor trigger chaining

The flow of execution from sensor to sensor is defined by one or more sensor trigger lists in the firmware. Trigger lists are calculated during the sensor interface initialization and are stored as a linked list using each sensor's `triggerList` pointer, with each physical sensor as the head of a sensor trigger list. The firmware first uses each virtual sensor's trigger source sensor type to locate its parent sensor. The virtual sensor is then added to the end of the trigger list which its parent is a member of. Trigger source and priority level values in virtual sensor descriptors can be used to create complex chains of sensor triggers. An example of a simple trigger list is shown in Figure 7.

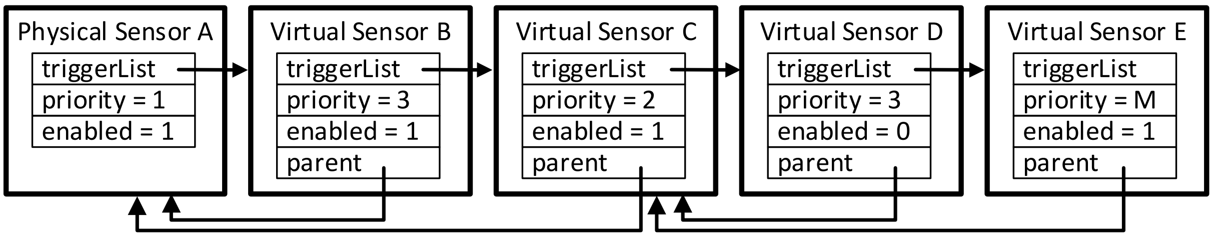


Figure 7. Example Trigger List

A trigger list may span multiple priority levels; however the list **cannot** be re-triggered until all sensor drivers on the list have finished executing. Alternatively, a secondary trigger list can be programmatically triggered to decouple the retriggering. As an example trigger list, a simple activity algorithm can be realized according to the following sequence: a timer triggers a virtual tilt, then the virtual tilt triggers a virtual activity. A more complex sensor dependency is shown in Figure 8. This example shows the trigger lists parsed by `stuffelf` at build time. Each sensor is introduced by its priority level in square brackets.

Note that if a triggered sensor cannot serve the request in time, e.g. due to priority or processor load, it is possible that data necessary for that sensor is overwritten and effectively lost. The data is transferred via a single data object, and no buffering in the form of a FIFO is applied.

```
--------- Trigger Lists (physical source) ---------
[1] accel sensor [DriverID 48]
[1] gyro sensor [DriverID 49]
[1] sigmot sensor [DriverID 44]
   [6] hw sigmot sensor [DriverID 239]
   [6] wakeup hw sigmot sensor [DriverID 181]
[1] stepcnt sensor [DriverID 47]
   [6] hw stepcnt sensor [DriverID 238]
   [6] wakeup hw stepcnt sensor [DriverID 211]
[1] stepdet sensor [DriverID 46]
   [6] hw stepdet sensor [DriverID 237]
   [6] wakeup hw stepdet sensor [DriverID 212]

--------- Trigger Lists (virtual: timer) ---------
25Hz Timer
  [2] hang detector sensor [DriverID 224]

--------- Trigger Lists (virtual: NO Source) ---------
unknown
  [3] BSX sensor [DriverID 240]
    [2] accel corr sensor [DriverID 241]
    [2] accel offset sensor [DriverID 209]
    [2] accel passthrough sensor [DriverID 205]
    [2] accel raw sensor [DriverID 203]
    [2] activity sensor [DriverID 235]
    [2] game rotvec sensor [DriverID 252]
    [2] glance status sensor [DriverID 234]
    [2] grav sensor [DriverID 247]
    [2] gyro corr sensor [DriverID 243]
    [2] gyro offset sensor [DriverID 208]
    [2] gyro passthrough sensor [DriverID 207]
    [2] gyro raw sensor [DriverID 244]
    [2] linaccel sensor [DriverID 246]
    [2] pickup status sensor [DriverID 233]
    [4] tilt sensor [DriverID 236]
    [2] wakeup accel corr sensor [DriverID 192]
    [2] wakeup accel raw sensor [DriverID 204]
    [2] wakeup game rotvec sensor [DriverID 200]
    [2] wake status sensor [DriverID 232]
    [2] wakeup grav sensor [DriverID 198]
    [2] wakeup gyro corr sensor [DriverID 194]
    [2] wakeup gyro raw sensor [DriverID 195]
    [2] wakeup linaccel sensor [DriverID 197]
```

Figure 8: Example of Complex Sensor Dependency

Note that while the sensor framework supports any type of physical sensor (e.g., `BSX_INPUT_ID_ANYMOTION`), only the `BSX_INPUT_ID_MAGNETICFIELD`, `BSX_INPUT_ID_ACCELERATION` and `BSX_INPUT_ID_ANGULARRATE` physical sensors are expected by the sensor fusion algorithm and have corresponding output sensors. All other physical sensors must have a corresponding virtual sensor that consumes the physical data and provides it to the host. This can be seen in Figure 8, where a virtual step counter driver is being triggered by the physical step count sensor in order to provide the output to the host.

4.1.5 Driver hang detection

The sensor framework includes a timer-based hang detector driver which monitors physical drivers for possible hangs. Hangs are detected if the driver does not start producing samples within 1 second of turning a sensor on or if the number of actual samples a physical driver produces is less than the expected number of samples, based on the current rate. The hang detector runs at a rate of 25 Hz.

In the event a hang is detected, the hang detector will issue a reset to the physical driver.

Hang detection does not run for a physical driver if any of the following conditions are true.

- The `no_hang` flag in the physical driver descriptor is set.
- The current power mode is SensorPowerModeInteruptMotion.
- The current rate is less than 3 Hz.

4.2 Drivers directory structure

Sensor driver code must be located in its own directory in the $SDK/drivers tree. The directory name should reflect the device name and driver type – for example `AK09915Mag`. Each driver directory should include four mandatory files as outlined in Table 8.

Table 8: Driver Directory Content

File in Driver Directory	Note
CMakeLists.txt	Compilation Makefile for driver source code
SensorNameType.c	Source code for driver functions
SensorNameType.h	Header file typically defining register locations and other constants for the driver

The developer is encouraged to use one of the available driver source codes as a template for creating new drivers.

4.3 Driver CMakeLists.txt file

Figure 9 shows an example of a driver *CMakeLists.txt* file. The *CMakeLists.txt* file automatically pulls in the sources from each driver. The user typically does not need to modify it. The driver directory name must be added to the `ENABLED_DRIVERS` definition in the appropriate *$SDK/common/config.7189_di03_*.cmake* file to be included in the SDK build.

`DRIVER_ID` is a unique value which can be queried by the Host to identify the driver ID of the sensor driver in the system. This allows more generic host driver code to be implemented for the BHy2. Currently unused driver IDs in the ranges of 100-125 and 165-180 can be used for new drivers. Note that a corresponding definition for the driver ID will be set by the build system when compiling the driver.

```
SET(DRIVER_ID 132)
get_filename_component( DRIVER_KEY ${CMAKE_CURRENT_LIST_DIR} NAME)

project(${DRIVER_KEY} C)

FILE(GLOB SOURCES "*.c")

include_directories(../../libs/BSXSupport/includes/
          ../../libs/BSX/includes/)

ADD_ARC_DRIVER(${DRIVER_KEY} {DRIVER_ID} ${SOURCES})
```

Figure 9: Custom Driver CMakeLists.txt Example

4.4 Checking for existing Driver IDs

There is a python script in the root directory of the SDK. Running it will show the existing driver names and associated driver IDs. Using this script will need an existing installation of Python.

```
$ python find_BHy2_driver_IDs.py
```

4.5 Writing driver code

The actual sensor driver code is typically written in two files. *SensorNameType.h* should contain the sensor register map and constant definitions. Its use is highly recommended for improved readability. The majority of the code is written in the *SensorNameType.c* source file. Its individual parts are described in the following subsections.

4.5.1 Recommended include files

Figure 10 shows an example of include files typically used for sensor driver code. *SensorAPI.h* defines constants and structures used in driver code and includes necessary sensor bus definitions. *Timer.h* allows the developer to use the timer for sensor access scheduling and is necessary for polling sensors. *bsx_support.h* provides access to BSX algorithm data.

```
#include <SensorAPI.h>          /* Interface available to custom
hooks and drivers */
#include <Timer.h>              /* Timer control routines */
#include <bsx_support.h>        /* Access to bsx algorithm data */
```

Figure 10: Driver Include Files Example

4.5.2 Sensor communication support

Most sensor devices are controlled by the BHy2 via a SPI or I2C interface. There are three possible sensor interface busses: SIF0, SIF1, and SIF2. SIF0 and SIF1 can be configured to use a SPI master or I2C master block in the BHy2. SIF2 is always I2C. Note that the BHA260 does not expose the M1 bus. At most, there can be two enabled SPI masters or two enabled I2C masters, so not all combinations are possible. The combination used for a given firmware image is specified in the board configuration file's sif_selection line. See section 2.2.1 for more details.

The BHy2 makes it easy for driver writers by hiding the differences between the I2C and SPI masters with a general purpose Sensor Bus design and associated read and write functions.

4.5.2.1 Sensor communication APIs

There are a variety of read and write functions to meet different needs. All the functions take a pointer to the device structure which is a member of the physical driver's sensor descriptor structure. The non-blocking functions also take a void pointer to a data parameter; this is a user-defined value which will be passed unmodified to the user's callback function as the second parameter. The function prototypes can be found in *$SDK/libs/SensorInterface/includes/SensorAPI.h.*

- Blocking – waits for the read or write to complete before resuming execution of the calling function
    - `SensorStatus write_data(const Device *device, UInt8 reg, UInt8 *buffer, UInt8 bytes)`
    - `SensorStatus read_data(const Device *device, UInt8 reg, UInt8 *buffer, UInt8 bytes)`
- Non-blocking – schedules the read or write and immediately returns to the calling function; when the read or write actually completes, the provided callback function is called; for writes, the buffer of data to write must continue to exist until the callback is called, which means it cannot be allocated on the stack of the calling function – it must be global.
    - `void write_data_nonblocking(const Device *device, UInt8 reg, UInt8 *buffer, UInt8 bytes, void (*callback_fn)(SensorStatus, void*), void *data)`
    - `void read_data_nonblocking(const Device *device, UInt8 reg, UInt8 *buffer, UInt8 bytes, void (*callback_fn)(SensorStatus, void*), void *data)`
- Inline Non-blocking – only available for writes. These calls can be used to send 4 or less bytes from a buffer local to the calling function which are used immediately – they do not need to persist until the callback, as in the other non-blocking writes. Note that `write_data_slow_inline_nonblocking` also introduces a delay between byte transfers (see "Slow" below).
    - `void write_data_inline_nonblocking(const Device *device, UInt8 reg, UInt8 *buffer, UInt8 bytes, void (*callback_fn)(SensorStatus, void*), void *data)`
    - `void write_bytes_inline_nonblocking(const Device *device, const UInt8 *buffer, UInt8 bytes, void (*callback_fn)(SensorStatus, void*),void* data)`
    - `void write_data_slow_inline_nonblocking(const Device *device, UInt8 reg, UInt8 *buffer, UInt8 bytes, void (*callback_fn)(SensorStatus, void*), void *data, float delay)`
- Slow – some sensor devices, especially when in low power states, require long delays between byte transfers; these slow variants take a delay parameter which sets the minimum time between bytes in milliseconds
    - `void write_data_slow_nonblocking(const Device *device, UInt8 reg, UInt8 *buffer, UInt8 bytes, void (*callback_fn)(SensorStatus, void*), void *data, float delay)`
    - `void read_data_slow_nonblocking(const Device *device, UInt8 reg, UInt8 *buffer, UInt8 bytes, void (*callback_fn)(SensorStatus, void*), void *data, float delay)`
- Register-less – the above APIs work well with sensor devices which use a single byte to specify which of the sensor's internal registers is to be read from or written to; for devices which cannot work in this mode, there are the following functions
    - `void write_bytes_nonblocking(const Device *device, UInt8 *buffer, UInt8 bytes, void (*callback_fn)(SensorStatus, void*), void *data)`
    - `void read_data_nonblocking(const Device *device, UInt8 *buffer, UInt8 bytes, void (*callback_fn)(SensorStatus, void*), void *data)`

4.5.2.2 Sensor communication best practices

The blocking read and write APIs are simple, but they can have a dramatic impact on the runtime behavior of the BHy2. Whenever these blocking APIs are called by a driver, any other driver running at the same interrupt priority level will be prevented from running even if they use a different SIF. This is especially bad if the driver using the blocking APIs uses slow I2C transfers while the blocked driver uses fast SPI transfers and is running at a much higher sample rate.

Another problem can occur if the driver uses delay functions between blocking transfers. These delay functions will again prevent any other physical driver at the same priority level from doing useful work.

It is best to write your driver in an event-driven, non-blocking fashion. This allows the BHy2 sensor framework to maximize performance of all sensors on all SIFs, and ensures any non-sensor interrupt handling can proceed efficiently.

Here are some helpful tips:

1. The driver initialization function is called when the system starts up AND later to recover from sensor failures; so it is important that all driver initialization functions use non-blocking calls and state machines, so they have minimal impact on the rest of the system
2. Do not use blocking calls; use non-blocking calls which handle next steps in a series of steps from one or more callback functions
3. The callback parameter can be NULL if no further processing must be done upon receipt of read data, but, your driver will not be able to detect fatal read failures on I2C devices without that callback
4. Do use the data parameter for each callback to simplify your logic; it can be used to maintain a state variable, so one callback function can handle a multi-step initialization function
5. Do not use fixed delays but instead use the slow APIs; these use the timer hardware so other drivers can proceed as needed until the timer expires
6. Do use the inline writes for small writes, so you do not have to use global buffers, which consume RAM unnecessarily
7. For non-blocking writes larger than 4 bytes or for non-blocking reads, use global buffers, as they must be valid in both the original calling function and later in the callback function.

4.5.3 Sensor descriptor structure

In order to achieve interoperability of application code with multiple different sensors from many vendors, the driver interface is defined in a sensor independent fashion using a sensor descriptor. Every sensor driver implementation must provide an initialized declaration of a sensor descriptor for that sensor. The sensor descriptor is composed of a common sensor descriptor header followed by data specific to physical or virtual sensors.

Note that a sensor driver must not directly modify data in its own sensor descriptor, aside from the members that are initialized at build time or initialization, with the exception of the `status.enabled` bit (see Table 9) and the `int_enabled` bit (see Table 15). Also, a sensor driver must not modify the sensor descriptor data for any other sensor drivers. The sensor framework is completely responsible for managing and setting all other sensor descriptor information. In addition, some sensor descriptor fields may be configured through parameter I/O by the host.

4.5.3.1 Sensor descriptor header

Physical and virtual sensors share a common header that includes the following fields.

- TriggerList
- SensorInfo
- SensorType

The **TriggerList** field is a pointer to a sensor descriptor header and is used to form a linked list of sensors that are triggered by a single sensor trigger source. During the build process, the `stuffelf` utility uses information from the trigger source field for all virtual sensors to form linked lists of sensors that are triggered by a single source sensor trigger. Driver code should **not** modify this list.

The **SensorInfo** structure outlined in Table 9 provides basic information about a virtual or physical sensor driver, as well as the current status of the driver. The definition of this structure can be found at *$SDK/libs/SensorInterface/includes/SensorAPI.h*. The driver ID and version are filled in by `stuffelf` based on information provided in the driver config file. The status fields are updated during runtime by the sensor framework and cannot be modified by the user with the exception of `status.enabled`. This bit can be modified by the user in the `VirtualSensorDetermined` hook to turn on a sensor that the user requires to be on (whether the host requires it or not). See section 4.8 for details on how to use hooks.

Table 9: SensorDescriptor – SensorInfo Structure

Field	**Type**	**Description**	**Driver access**
`Id`	`UInt8`	The current driver ID of the sensor, should be set to DRIVER_ID. Stuffelf will incorporate the value set in the driver config file at build time.	Set to DRIVER_ID in descriptor definition, then read-only
`version`	`UInt8`	The current driver revision of the sensor, should be set to DRIVER_REV. Stuffelf will incorporate the value set in the driver config file at build time.	Set to DRIVER_REV in descriptor definition, then read-only
`resolution`	`UInt8`	Number of bits of resolution of the sensor input data	Set in descriptor definition, then read-only
`reserved`	`UInt8`	Reserved byte	None
`status.triggered`	`UInt8`	The sensor has been scheduled to run	Read-only via the `isSensorTriggered` API function
`status.enabled`	`UInt8`	The sensor has been enabled	Can be modified only in the `VirtualSensorDetermined` hook
`status.i2c_nack`	`UInt8`	The sensor has had an I2C communication error	Read-only
`status.devid_error`	`UInt8`	The sensor was not detected at the supplied address	Read-only
`status.transient_error`	`UInt8`	The sensor had a transient error	Read-only
`status.list_ran`	`UInt8`	Flag used to track whether all triggered sensors on a trigger source's trigger list have run.	Read-only
`status.rate_changed`	`UInt8`	The host interface has updated the lastRequestedRate entry	Read-only
`status.range_changed`	`UInt8`	The host interface has updated the lastRequestedRange entry	Read-only
`status.pending_trigger`	`UInt8`	The sensors trigger has not finished running, but a new trigger has arrived	Read-only
`status.enabled_ack`	`UInt8`	The sensor framework has acked that the driver is enabled	Read-only
`status.just_turned_on`	`UInt8`	Signal from physical sensor to associated virtual sensor; used to trigger an on-change sensor to output its current value	Read-only

Field	Type	Description	Driver access
`status.reserved`	`UInt8`	Reserved	None
`pad0`	`UInt32`	Reserved	None

The **SensorType** structure outlined in Table 10 provides basic sensor properties, including the sensor ID, sensor type and a number of flags that specify the sensor's behavior. The definition of this structure can be found at *$SDK/libs/SensorInterface/includes/SensorAPI.h*. These values should be set in the declaration of the driver's sensor descriptor, and should **not** be modified during run-time.

Table 10: SensorDescriptor – SensorType Structure

Field	Type	Description
`value`	`UInt8`	Sensor ID
`flags`	`2 bits`	Sensor type flags - Virtual, Timer or Physical
`wakeup_ap`	`0 / 1`	If true, sensor data is placed in the wakeup FIFO, otherwise the non-wake FIFO.
`no_hang`	`0 / 1`	Disable hang detection code for physical sensors
`no_decimation`	`0 / 1`	Always use the parent sensor rate, even if a slower rate is requested by the host
`on_change`	`0 / 1`	This is an on-change sensor, which will require special processing when samples are lost
`always_on`	`0 / 1`	This is used to ensure that the driver is always enabled, even if no children are enabled
`hidden`	`0 / 1`	This sensor should be hidden by the host interface. Note that sensors may also be made invisible by assigning a sensor type in the invisible range.
`decimate_integer`	`0 / 1`	If 0, virtual sensor output rate will decimate from the source sensor rate by a power of two. If 1, any integer value will be used.
`on_change_map_bit`	`5 bits`	Bit map value assigned to each on-change sensor, used to track whether an on-change sensor has been triggered. Used when handling lost on change events.
`reserved_flags`	`2 bits`	Reserved
`reserved`	`UInt8`	Reserved

4.5.3.2 Physical sensor descriptor fields

Following the sensor descriptor header, the physical sensor descriptor has a number of additional fields specific to physical sensors, including the following.

- Device structure
- Function pointers
- Data parameters

The **Device structure** outlined in Table 11 provides a physical descriptor of a physical sensor device. Note that some device data fields of physical sensors are replaced by the `stuffelf` utility using data in the board configuration file, since their value depends on the actual application (`options.i2c.address, irqPin, irqDis`). The I2C and SPI protocol parameters are used to configure the physical interface to the sensor.

Table 11: SensorDescriptor – Physical Device Structure

Field	Value Type	Description
`interface.handle`	`void*`	The interface handle based on the interface type, initialized by the firmware
`interface.type`	`SensorInterfaceType`	The selected interface type for the sensor
`interface.options`	`SensorInterfaceType`	All possible interface types for this sensor driver
`options.i2c`	`I2C_Device_t`	See Table 12
`options.spi`	`SPIM_Device_t`	See Table 13
`irqPin`	`5 bit`	GPIO Pin to monitor for interrupts
`irqEdge`	`0/1`	1=Rising edge, 0=Falling edge
`irqDis`	`0/1`	1=No GPIO pins used for this sensor

Table 12: I2C Device (I2C_Device_t)

Field	Value Type	Recommended Value	Notes
`maxClock`	`10 bits`	100, 400, 1000	Max clock speed supported by the device (kHz)
`Reserved`	`6 bits`		Reserved
`hasClockStretching`	`0 / 1`	0	0=No clock stretching, 1=Clock stretching
`address`	`7 bits`		7-bit slave address (MSB = 0)
`Reserved`	`UInt8`		Reserved

Table 13: SPI Master Device (SPIM_Device_t)

Field	Value Type	Notes
`maxClock`	`UInt16`	Max SPI clock speed supported by the device (kHz)
`csPin`	`UInt8`	GPIO pin number of the chip select signal
`csLevel`	`UInt8`	Chip select value to select the chip, 1=Active high, 0=Active low
`cpol`	`0 / 1`	SPI clock active polarity, 0=negative, 1=positive
`cpha`	`0 / 1`	SPI clock active phase, 0=leading clock edge, 1=trailing clock edge
`en3wire`	`0 / 1`	Disable/enable 3-wire SPI
`lsb_first`	`0 / 1`	SPI data shift direction, 0=MSB first, 1=LSB first
`Reserved`	`4 bits`	Reserved
`reg_shift`	`3 bits`	Number of bits to shift the register address by to generate the command byte
`read_pol`	`0 / 1`	Read signified by a 0 or a 1 in the command byte
`read_bit`	`4 bits`	Bit position for the read/write bit, often bit 7 or bit 0.

The **function pointers** in a physical sensor descriptor are outlined in Table 14. These fields specify pointers to functions that set and retrieve sensor state and parameter settings as well as scheduling a read of the sensor sample. These values should be set in the declaration of the driver's sensor descriptor, and should **not** be modified during run-time. More details on each of these functions can be found in section 4.4.4.

Table 14: SensorDescriptor – Physical Sensor Function Pointer Fields

Function	**Description**
Sensor State and Parameter Setting	
`initialize`	Verify the sensor connection and initializes the sensor into a known power down state
`set_power_mode`	Put the sensor to the requested power mode
`set_sample_rate`	Put the sensor to the requested rate to produce data (sample rate)
`set_dynamic_range`	Set the sensor to the requested dynamic range, return the actual dynamic range
`set_sensor_ctl`	Set specific sensor configuration (FOC, OIS, or FST)
`enable_interrupts`	Enable interrupt generation by sensor
`disable_interrupts`	Disable interrupt generation by sensor
Sensor State and Parameter Query	
`get_power_mode`	Return the current sensor power mode
`get_sample_rate`	Return the current sensor sample rate
`get_scale_factor`	Return multiplicative scale factor for conversion of sensor result to common units
`get_dynamic_range`	Return the current dynamic range sample rate
`get_sensor_ctl`	Return specific sensor configuration (FOC, OIS, or FST)
Sample Data Handling	
`get_sample_data`	Schedule I2C read transaction to read a sensor sample

The **Data** parameters in a physical sensor descriptor are outlined in Table 15. These data fields contain actual parameters which are needed for the system to handle physical sensors properly. The `CalMatrix` field is provided by `stuffelf` from the driver config file and is used to rotate (if needed) the physical sensor data to match the required sensor orientation of the device from a user's perspective. The `lastRequestedRate` and `lastRequestedRange` fields are used by the host interface API call for the requested rate/range if a rate/range change is needed.

Table 15: SensorDescriptor – Physical Sensor Data Fields

SensorDescriptor Item	**Value Type**	**Notes**
Data timestamp for the sensor		
`latestTimestamp`	`UInt64`	Latest valid timestamp
`newTimestamp`	`UInt64`	Latest timestamp, even for invalid data
Minimal value of parameters		
`minRate`	`float`	Minimal rate (in Hz) of the sensor
Maximal value of parameters		
`maxCurrent`	`float`	Maximal current draw of the sensor. units: [mA]
`maxRate`	`float`	Maximal rate (in Hz) of the sensor

SensorDescriptor Item	Value Type	Notes
`maxDynamicRange`	`UInt16`	Maximal dynamic range of the sensors in [units]
Default value of parameters		
`defaultDynamicRange`	`UIn16`	Default dynamic range to use if not requested by the host
Request for parameter changing		
`lastRequestedRate`	`float`	Last requested rate for the sensor as determined by the outerloop. Do not modify.
`lastRequestedRange`	`UInt16`	Last requested range for the sensor. Do not modify.
Sensor calibration		
`CalMatrix`	`SInt8 [9]`	3-axis calibration matrix to apply to 3-axis sensor data
Data access		
`sensorData`	`void*`	Direct access to sensor data
`numAxis`	`7 bits`	Number of axes in the sensor
Hang detection		
`resetDivisorCount`	`UInt8`	Number of sample events needed before a sensor is considered alive
`resetDivisorTimeout`	`UInt8`	Number of 25 Hz timer intervals to wait before checking for a hung sensor
`sampleCount`	`UInt8`	Number of sample events since the last hang check
`resetCount`	`UInt8`	Number of recent reset events for the sensor
`resetCountLimit`	`UInt8`	
`needsReset`	`UInt8`	Flag indicating that the sensor should be reset
`int_enabled`	`0 / 1`	Sensor interrupt or timer state, 0 = disabled, 1 = enabled. Should be set/cleared when the driver enables/disables the sensor interrupt.
Sensor control parameter read/write		
`sensorControlCode`	`7 bits`	Sensor control parameter code
`sensorControlDir`	`0 / 1`	Sensor control parameter direction, 0 = write, 1 = read

4.5.3.3 Virtual/timer sensor descriptor fields

Virtual and Timer sensors share the same Sensor Descriptor. For a virtual/timer sensor, the Sensor Descriptor is composed of a Sensor Descriptor Header followed by a number of additional fields specific to virtual/timer sensors, including the following categories.

- Trigger source
- Physical source
- Function pointers
- Data parameters

The **Trigger source** structure specifies the ID (value) and type (flags) of the sensor that is a trigger for this virtual sensor. For example, a virtual accel driver would specify its trigger source value as `BSX_INPUT_ID_ACCEL` and its trigger source flags as `DRIVER_TYPE_PHYSICAL_FLAG`. A timer sensor must also fill in the timer field in the trigger source structure. At build time, the `stuffelf` utility uses the trigger source from each virtual sensor to compile the trigger lists. Trigger source data of virtual sensors are set during initialization.

Table 16: Trigger Source Timer

SensorDescriptor Item	Value Type	Notes
`Decimate`	`2 bits`	0=Never, 1=Always, 2=Power of two
`decimate_max`	`6 bits`	Maximal value for rate decimation
`Index`	`SInt8`	
`Rate`	`UInt16`	Actual sensor data rate

The **Physical source** specifies the ID and type of the physical sensor that primarily affects this virtual sensor's data.

The **Function pointers** in a virtual sensor descriptor are outlined in Table 17. These fields specify pointers to functions that process and allow access to sensor data as well as initialize the virtual sensor. More details on each of these functions can be found in section 4.4.4.

Table 17: SensorDescriptor – Virtual Sensor Function Pointer Fields

Function	Description
Sensor State and Parameter Setting	
`Initialize`	Verify sensor connection and initializes sensor into a known power down state
Sample Data Handling	
`handle_sensor_data`	New data processing
`get_last_sensor_data`	Used for on-change sensors

The **Data** parameters in a virtual sensor descriptor are outlined in Table 18. Virtual Sensor Descriptor data fields contain actual parameters which are needed for the system to handle sensors properly. If the programmer wishes to update the dynamic range of the sensor, they should use the host interface API call to update the requested rate/range if a rate/range change is needed.

Table 18: SensorDescriptor – Virtual Sensor Data Fields

SensorDescriptor Item	Value Type	Notes
`priority`	`UInt8`	The priority level to run the virtual sensor calculations
`decimationLimit`	`UInt8`	Internal use only
`decimationCount`	`UInt8`	Internal use only
`outputPacketSize`	`UInt8`	The number of bytes in the sensor output packet, **excluding** the Sensor ID
`timestamp`	`UInt64`	The timestamp of the source causing the driver to be triggered
Request for parameter changing		
`lastRequestedRange`	`UInt16`	The last requested range for the sensor as determined by the host interface. Do not modify.
`lastRequestedRate`	`float`	The last requested rate for the sensor as determined by the host interface. Do not modify.

4.5.4 Sensor driver functions

This section discusses the functions that should be defined for each sensor driver.

Not all functions from function pointer fields for physical and virtual sensors must be implemented. Please see section 4.6 for the functions required. Unimplemented function pointers must be set to `NULL` in the Sensor Descriptor data structure.

Note that it is very important that none of the sensor descriptor functions should be called directly from driver code. They are provided by each driver, but must only be called by the sensor framework.

4.5.4.1 Sensor State and Parameter Setting Functions

The following two sections describe individual sensor functions defined in the Sensor Descriptor.

- **`initialize`** is called whenever the BHy2 transits from `Initialized` state to operational state. This function may also be called if a sensor is in an unknown/unusable state. This function **shall not** affect the state of another sensor driver. In the event of a composite sensor, this driver **may** temporarily affect the state of another sensor driver, however once initialization is complete, the sensor should return to the state specified by the other sensor driver. This function **must** perform the following operations:
  - o Verify that a device is found with the specified I2C/SPI configuration. The status code `SensorErrorNonExistant` must be returned if no device is found.
  - o Verify that the driver is capable of talking with the found device. This is often done by checking the `WHO_AM_I` register if available. If a device that the driver does not understand is found, `SensorErrorUnexpectedDevice` must be returned by the function.
- Once a known device is found, the initialize function should reset the sensor to a known state for the driver, as well as ensure that the device is in the `SensorPowerModePowerDown` power mode.
  `set_power_mode` is called shortly after `initialize` to handle sensor power mode transitions. It is also called during sensor teardown operation before transition to `Initialized`. This function **shall not** affect the power mode of other sensor drivers. Supported sensor power modes are defined in `SensorAPI.h` and are described in Table 19. Requirements for which power modes are required to be implemented are shown in Table 20. The driver developer should map sensor power modes to actual power modes of a particular sensor (see example drivers for inspiration). This function **must** return the actual power mode selected by the sensor driver, even if the requested power mode is unsupported. After setting the required power mode, the callback function `sensorPowerModeChanged` must be called to report that the power mode has been updated.

Table 19: SensorPowerMode Definition

SensorPowerMode	Notes
`SensorPowerModePowerDown`	Lowest power state supported by the sensor driver. Sensor data conversions should be disabled here and the device should be shut down. The driver should be able to transition to an active state within **1000 ms.**
`SensorPowerModeSuspend`	Low power state where sensor data conversions have been disabled, however the system must be able to transition into an active state within **100 ms.** This state may be the same as `SensorPowerModePowerDown`.
`SensorPowerSelfTest`	Perform a sensor self-test. The sensor driver must call the `reportSelfTestResult` function once the test has completed, as well as transition to the `SensorPowerModePowerDown` state. Note that the x, y, z offset results should be provided in the native units for a given sensor, e.g., mg for accel or dps for gyroscope.
`SensorPowerModeFOC`	Enter a FOC calibration mode in the sensor. The sensor driver must call the `reportFOCResults` function once the test has completed, as well as transition to the `SensorPowerModePowerDown` mode once the FOC is complete. Note that the x, y, z offset results should be provided in the native units for a given sensor, e.g., mg for accel or dps for gyroscope.
`SensorPowerModeInterruptMotion`	Put the sensor to a special state where the device only provides interrupts when a specified sensor data threshold has occurred.

SensorPowerMode	Notes
`SensorPowerModeOneShot`	Cause the sensor to complete a single sensor data conversion. `SensorPowerModeSuspend` or lower should be entered automatically afterwards.
`SensorPowerModeLowPowerActive`	Operational power state where sensor settings are optimized for low power. This state may have higher noise than the `SensorPowerModeActive` mode. If `SensorPowerModeActive` is implemented, this state must also be implemented; however it may be the same state as `SensorPowerModeActive`.
`SensorPowerModeActive`	Operational power state where sensor settings are optimized for high performance. This state should have minimal noise in sensor measurements.

Table 20: set_power_mode Driver Requirements

Power Mode	Sensor Type/Requirements	
	All Physical Sensors	Accelerometer
`SensorPowerModePowerDown`	Required - may duplicate `SensorPowerModeSuspend`	Required - may duplicate `SensorPowerModeSuspend`
`SensorPowerModeSuspend`	Required	Required
`SensorPowerModeSelfTest`	Optional	Optional
`SensorPowerModeFOC`	Optional	Optional
`SensorPowerModeInterruptMotion`	Optional	Recommended, will be required in future releases
`SensorPowerModeOneShot`	Optional	Optional
`SensorPowerModeLowPowerActive`	Required - may duplicate `SensorPowerModeActive`	Required - may duplicate `SensorPowerModeActive`
`SensorPowerModeActive`	Required	Required

- **`set_sample_rate`** is used to set the sensor measurement rate. This function is called after initialization and any time the requested sensor sample rate has changed. If a sensor supports a continuous conversion mode, this function will set the corresponding register in the sensor to enable the mode. The function typically checks an internal list of supported rates and sets the one which is greater or equal to the requested rate.
  In the case of **polled sensors,** the requested rate is ensured by the internal timer. In that case **`set_sample_rate`** uses the timer API defined in `Timer.h` for scheduling the sensor interrupt at regular intervals (For an example of a polled sensor driver refer to the `AK09915Mag`). Note that `set_sample_rate` and `set_power_mode` can be called in any order. Actual operation of the sensor should be controlled by `set_power_mode` while the `set_sample_rate` should only set the rate register.
  This function is required if the sensor supports `SensorPowerModeLowPowerActive` or `SensorPowerModeActive`. This function **shall not** modify the sample rate of another sensor driver **unless** the sensor is a composite sensor, in which case, the sample rate of the other composite sensor driver may be updated. The composite sensor rate **should be** greater than or equal to all requested rates for the sensor if they cannot be set individually. All sensors **must** implement the callback function `sensorRateChanged()` that updates variables for hang detection calculation.
- **`set_dynamic_range`** is used to change the maximum range of a sensor. Implementation is **required**, even if it is an empty function. This function is called during initialization to set the sensor into the desired range for the main fusion algorithm. This function **must** set the sensor to the specified range or higher. For example, if a gyro supports 100, 500, 2000 dps, a request of 300dps will result in a range of 500 dps, while a value of 1000 dps would result in a value of 2000 dps being set.
- **`enable_interrupts / disable_interrupts`** should enable/disable interrupts for a given sensor at its source (the sensor). Implementation is **optional**. In some sensors this is equivalent to enabling/disabling actual operation. If that is

the case, the driver should check the current driver power state and enable operation only when sensor is in one of its operational states.

- In case of **polled sensors**, the enable/disable interrupt functions should enable/disable scheduling the timer interrupt (refer to the `AK09915MagSensor` driver code for an example). The enable interrupts function should allow interrupts from the sensor to be received within 100 ms of the function being called.

4.5.4.2 *Sensor state and parameter query functions*

Implementation of all query functions is **required** for all sensor drivers.

- `get_power_mode, get_sample_rate,` and `get_dynamic_range` should return the actual sensor settings. Note that the driver should cache power mode, sample rate value and dynamic range values in internal state variables instead of reading them from the sensor over the sensor bus.
- `get_scale_factor` is used to convert raw sensor data to calibrated values. It returns factors specific to a given sensor type. Note that these values could depend on the dynamic range setting.

4.5.4.3 *Sensor data handling functions*

Table 21 provides a summary of the sensor data handling functions for both physical and virtual sensors.

Table 21: Summary of Sensor Data Handling Functions

Sensor Driver Function	Physical	Virtual
`interrupt_handler`	`sensorInterruptHandler`	N/A
`get_sample_data`	Save `callback` Read sensor data using sensor bus	N/A
`handle_sensor_data`	Extract data into correct form (from physical sensor data format to format expected by virtual handle_sensor_data function) Save data to `self->sensorData` Call saved `callback`	Begin virtual calculations. Use `reportSensorEvent` to cause data output

Figure 11 shows the interaction between the Sensor Driver routines for physical sensors and other software elements. The physical sensors feed data to virtual sensors using the sensor framework. The programmer should refrain from including calculations in physical sensor drivers, and should instead utilize virtual sensor drivers for calculations.

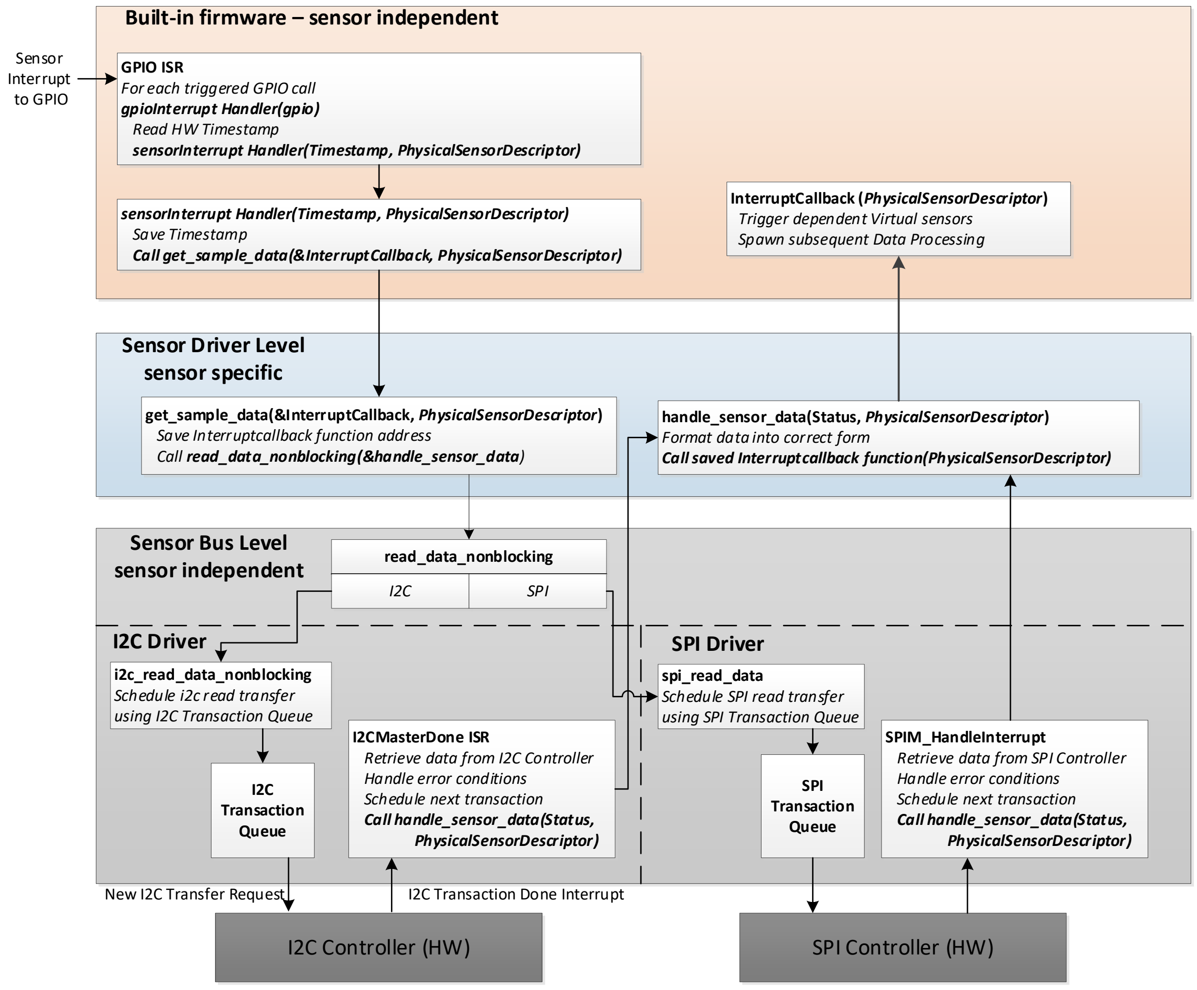

Built-in firmware – sensor independent
Sensor Interrupt to GPIO
GPIO ISR
For each triggered GPIO call
gpioInterrupt Handler(gpio)
Read HW Timestamp
sensorInterrupt Handler(Timestamp, PhysicalSensorDescriptor)
sensorInterrupt Handler(Timestamp, PhysicalSensorDescriptor)
Save Timestamp
Call get_sample_data(&InterruptCallback, PhysicalSensorDescriptor)
InterruptCallback (PhysicalSensorDescriptor)
Trigger dependent Virtual sensors
Spawn subsequent Data Processing
Sensor Driver Level sensor specific
get_sample_data(&InterruptCallback, PhysicalSensorDescriptor)
Save Interruptcallback function address
Call read_data_nonblocking(&handle_sensor_data)
handle_sensor_data(Status, PhysicalSensorDescriptor)
Format data into correct form
Call saved Interruptcallback function(PhysicalSensorDescriptor)
Sensor Bus Level sensor independent
read_data_nonblocking
I2C
SPI
I2C Driver
i2c_read_data_nonblocking
Schedule i2c read transfer using I2C Transaction Queue
I2C Transaction Queue
I2CMasterDone ISR
Retrieve data from I2C Controller
Handle error conditions
Schedule next transaction
Call handle_sensor_data(Status, PhysicalSensorDescriptor)
SPI Driver
spi_read_data
Schedule SPI read transfer using SPI Transaction Queue
SPI Transaction Queue
SPIM_HandleInterrupt
Retrieve data from SPI Controller
Handle error conditions
Schedule next transaction
Call handle_sensor_data(Status, PhysicalSensorDescriptor)
New I2C Transfer Request
I2C Transaction Done Interrupt
I2C Controller (HW)
SPI Controller (HW)

Figure 11: Physical Sensor Data Handling

A physical sensor signals availability of new data using a GPIO interrupt line. The GPIO interrupt service routine (ISR) determines which GPIO interrupts have fired and calls `gpioInterruptHandler` for each GPIO that has fired. `gpioInterruptHandler` is shared by all GPIO-based physical sensors. This function reads timestamps from the appropriate register and calls `sensorInterruptHandler` with the determined 64-bit timestamp and physical sensor descriptor.

`sensorInterruptHandler` stores the timestamp in appropriate internal data structures. It then calls the sensor specific `get_sample_data` function from the passed in physical sensor descriptor with an interrupt callback that is used to notify the framework once the new data has been read in.

The sensor specific `get_sample_data` typically calls `read_data_nonblocking` to read out the sensor data that caused the interrupt. This function must ensure that the passed in callback function is called once the sensor data has been read in. Typically, this requirement is fulfilled by storing the pointer in an internal static variable (part of the driver code) and calling it from the sensor bus read data callback function. This read data callback function is used to format the sensor data in a generic way, then to call the framework callback function.

`read_data_nonblocking` is part of the Sensor Bus Interface API that abstracts the lower level I2C /SPI bus interface from the sensor driver. It calls i2c_read_data_nonblocking or spi_read_data depending on whether the sensor driver initialized the sensor device as an I2C or SPI device. These functions schedule a read transfer on the bus that will be executed once all previous transactions have finished. If no pending transactions exist, then it will immediately start the requested transaction. Once the read transfer has completed, it will call back into the driver with the transaction result. Please note that the blocking version of `read_data/write_data` **cannot** be used in priority level 1 code (such as sensor bus callbacks, timer callbacks, parameter read/write handlers, and GPIO handlers).

The I2C /SPI callback functions extract the data result and status information from the data read on the bus. Typically, byte swapping, shifting, and basic calibrations are done here. The driver should also check the data validity based on sensor error indications (if available). Finally, these functions call back into the sensor framework utilizing the stored `interruptCallback` function. The `interruptCallback` triggers other virtual sensors that are dependent on this physical sensor.

4.5.4.4 Virtual sensor data handling functions

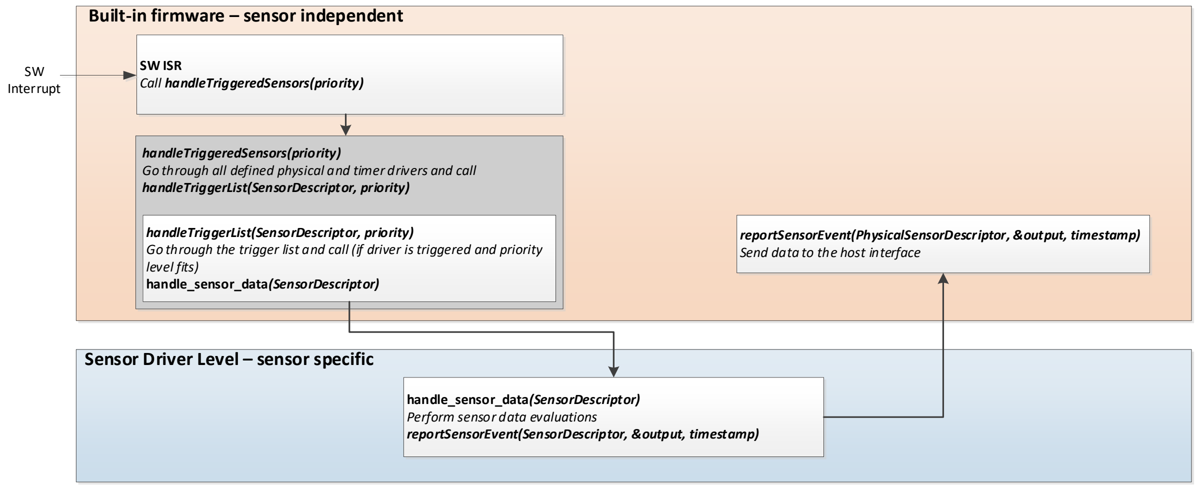

Built-in firmware – sensor independent
SW Interrupt
SW ISR
Call handleTriggeredSensors(priority)
handleTriggeredSensors(priority)
Go through all defined physical and timer drivers and call
handleTriggerList(SensorDescriptor, priority)
handleTriggerList(SensorDescriptor, priority)
Go through the trigger list and call (if driver is triggered and priority level fits)
handle_sensor_data(SensorDescriptor)
reportSensorEvent(PhysicalSensorDescriptor, &output, timestamp)
Send data to the host interface
Sensor Driver Level – sensor specific
handle_sensor_data(SensorDescriptor)
Perform sensor data evaluations
reportSensorEvent(SensorDescriptor, &output, timestamp)

Figure 12: Virtual Sensor Data Handling

Figure 12 shows the interaction between Virtual Sensor Driver routines and other software elements for virtual sensors. Virtual sensor drivers are able to consume data from physical sensors and other virtual sensors and also provide it to other virtual sensors. These drivers should contain any needed calculations. The following paragraphs describe the typical flow of control and data when virtual sensors are triggered.

Triggering of a virtual sensor is signaled by a software interrupt setting the `SWI2` bit to one in the `AR_SOFTWARE_INT` register. The SWI ISR calls the `handleTriggeredSensors` function as soon as the execution of a higher interrupt priority is finished.

`handleTriggeredSensors` goes through all defined physical drivers and timer drivers and calls `handleTriggerList` with input parameters referencing the `SensorDescriptorHeader* source` and `UInt8 priority` (the priority is passed in from SWI ISR routines).

`handleTriggerList` checks whether the input sensor is triggered. If it is, the function goes through the trigger list and calls `handle_sensor_data` functions for the sensors which are triggered and whose priority fits the required priority level.

`handle_sensor_data` virtual sensor function processes data and calls `reportSensorEvent` for sending data to the host interface when required.

4.5.5 Using custom sensor IDs to send data to the host

If a new custom virtual driver produces data that does not conform to an existing BSX sensor type already defined in the datasheets of BHI260AB and BHA260AB, Reference 1 or Reference 2, then you will need to define your own by completing the following steps.

1. Select a new Sensor ID (see section 4.1.2.4).
2. Set the virtual sensor descriptor `.type.value` field to the new Sensor ID.
3. Determine the data packet format (e.g., 3 16 bit signed integers).
4. Determine the size of the whole sensor data packet (1 byte for Sensor ID plus the size of the data packet). This value goes in the `.outputPacketSize` field of the sensor descriptor.

When you add a custom virtual sensor, you specify the new Sensor ID in the virtual sensor descriptor's `.type.value` field.

4.5.6 Connection between Driver ID and Sensor ID

The link between driver ID and sensor ID is made using the *CMakeLists.txt* file in the source directory for your new virtual driver, the board config file used to specify the contents of your firmware image, and the virtual sensor descriptor in the driver's .c file. The driver ID must be placed in the sensor descriptor in the .info.id field in addition to the *CMakeLists.txt* file.

The sensor ID only appears in the virtual sensor descriptor; only virtual sensors can output data to the host. If you add a new physical sensor driver (e.g., a humidity sensor), you also need to add a corresponding virtual sensor to be triggered by your physical sensor, access the physical sensor data, convert it to the proper host format, then send it to the FIFO with `reportSensorEvent`.

4.5.7 Virtual sensor host interface

When the firmware starts up, internal data structures used to organize the sensor drivers are initialized. These structures are then automatically used by the host interface firmware to handle host requests. This will be done for you; there is no effort to have your sensor driver configured or report its status.

These internal data structures exist for all non-wakeup and wakeup sensors. These structures include the current host request for each sensor's sample rate and latency; the size in bytes of the event to be placed in the FIFO; whether the sensor is on-change, etc. When your virtual sensor driver calls `reportSensorEvent`, the sensor framework uses these structures to decide which FIFO (none, either, or both) to insert the data into. If the sample rate the host requests for one FIFO is different from the other FIFO, the firmware will automatically decimate it so that fewer samples go into the lower rate FIFO.

4.5.8 Handling special cases

Table 21 summarizes the approach to write sensor drivers for the most typical sensors. Only the `get_sample_data / handle_sensor_data` functions have to be programmed and the interrupt handler has to be correctly selected. However, there are special cases which require more custom programing. Special cases are described briefly in Table 22. A detailed description of these special cases is beyond the scope of this document. The user is encouraged to contact Bosch Sensortec for further assistance with writing drivers for special cases.

Table 22: Summary Special Sensor cases

Special Case	What to do ?
Composite device with shared interrupt pin between multiple sensor types	• Custom `interrupt_handler` • Sensor bus read to determine source of interrupt • Sensor bus read callback function to direct execution to the generic interrupt handler for other sensor drivers (`sensorInterruptHandler()`)
Polling sensor with scheduled `start of measurement`	• Scheduling done using *Timer.h* functions • Timer ISR sends sensor bus command to start measurement • Sensor data availability signalled using GPIO and processing using standard interrupt handlers (gyro/mag/accel/sensor).
Polling sensor with scheduled `new sensor data read` (no sensor GPIO IRQ)	• No interrupt_handler (no GPIO interrupt) • Scheduling of data read done using `Timer.h` functions • Timer ISR must create timestamp and call standard interrupt handler (`sensorInterruptHandler()`)
Virtual sensor with continuous triggering	• Trigger source is Timer with a 0Hz rate • Cannot have any children
Virtual sensor with programmatic triggering	• Programmer must call `triggerSensors(descriptor)` to start the trigger chain • Rate is determined by programmer and may vary

4.6 Sensor data injection drivers

The Set Sensor Data Injection Mode and Inject Sensor Data commands give the host the ability to inject sensor data into the sensor framework instead of receiving sensor data from physical sensors. Sensor data injection requires that special data injection drivers are built into the firmware image instead of the normal sensor drivers. These drivers must register themselves with the sensor data injection module during initialization. The function prototype for this is shown in Figure 13. In addition, the driver must handle injected data in its `get_sample_data` function and pass the injected data on to the sensor framework.

```
SensorStatus SDI_initializeSensor(sdi_entry_t *entry);
```

Figure 13: Sensor Data Injection Function APIs

The following sections provide more details on sensor data injection driver requirements.

4.6.1 Initialization

Every sensor data injection driver must register information with the sensor data injection module in its initialization function. This information includes a pointer to its physical sensor descriptor, a pointer to the data buffer where incoming sensor data should be copied, and the packet size for incoming data. The driver should initialize an `sdi_entry_t` structure as shown in Figure 14 with this data and then call `SDI_initializeSensor`.

```
typedef struct _sdi_entry_type_
{
  PhysicalSensorDescriptor * sensor;
  UInt8 *dataBuffer;
  UInt8 packetSize;
  struct _sdi_entry_type_ *next;
} sdi_entry_t;
```

Figure 14: Sensor Data Injection Structure for Initialization

4.6.2 Set sample rate

The `set_sample_rate` function in a sensor data injection driver must inform the sensor framework of a sample rate change by calling `sensorRateChanged`. The sensor data injection module will then send an Injected Sensor Configuration Request to the host to inform the host of the rate change. The host should respond by adjusting the rate at which it is injecting data to the sensor.

4.6.3 Get sample data

When injected sensor data is received from the host the sensor data injection module will find the registered sensor driver with a matching driver ID and then copy the data into the driver's data buffer that was registered during the sensor driver initialization. It will then call the sensor interrupt handler for the sensor driver. This results in a call to the `get_sample_data` function for the sensor driver. At this point, the driver should perform any necessary calculations on the injected data, copy it into its `sensorData` buffer, and call the callback. The callback will perform the steps to insert the injected sensor data into the virtual sensor framework.

4.6.4 Other required sensor functions

Data injection sensor drivers must implement and provide emulated functionality for a minimum set of additional driver functions, including the following.

- `set_power_mode`
- `get_power_mode`

- `set_sample_rate`
- `get_sample_rate`
- `get_scale_factor` (acceleration, magnetic field, gyroscope, pressure, humidity, and temperature)
- `get_dynamic_range`
- `set_dynamic_range` (acceleration and gyroscope)

4.6.5 Driver config file and custom board file

The driver config file for a sensor data injection driver should specify the driver type as the intended physical driver type (e.g., accel, gyro, mag) and the driver ID as any allowed unused driver ID.

It is recommended that a new board file is used to include only the sensor data injection drivers as the physical drivers. Any virtual sensors that use the same type of physical source data may be included. When specifying the sensor injection driver as a physical sensor, the bus must be specified as `none` instead of SPI or I2C. If appropriate, the calibration and offset values should be identical to those of the actual physical sensor that is being emulated.

4.7 Driver coding requirements

All published sensor drivers must follow the following requirements:

- Req 1.1 The physical driver **must** include a `PhysicalSensorDescriptor` definition in the `.phys_sensor_descriptor` section.
- Req 1.2 The SensorDescriptor **must** contain a non-zero sensor `type`.
- Req 1.3 The SensorDescriptor **must** contain a non-zero sensor `maxI2CSensorSpeed`. (Ignored for SPI interfaces.)
- Req 1.4 The SensorDescriptor **must** contain a non-zero sensor `driverID`.
- Req 1.5 The SensorDescriptor **must** contain a non-zero sensor `driverVersion`.

- Req 2.1 The **`initialize`** function **must** be implemented for `BSX_INPUT_ID_MAGNETICFIELD`, `BSX_INPUT_ID_ACCELERATION`, and `BSX_INPUT_ID_ANGULARRATE`.
- Req 2.2 The **`initialize`** function **must** be implemented for all sensors supporting `SensorPowerModeLowPowerActive` or `SensorPowerModeActive`.
- Req 2.3 The **`initialize`** function **must** return `SensorErrorNonExistant` **if** no device was found at the specified I^{2}C address. (Ignored for SPI interfaces.)
- Req 2.4 The **`initialize`** function **must** return `SensorErrorUnexpectedDevice` **if** an unknown device was found at the specified I^{2}C address. (Ignored for SPI interfaces.)
- Req 2.5 The **`initialize`** function **must** place the sensor into the `SensorPowerModePowerDown` power mode.
- Req 2.6 The **`initialize`** function **must not** modify the power mode of any other sensor driver.

- Req 3.1 The **`set_power_mode`** function **must** be implemented for all sensors.
- Req 3.2 The **`set_power_mode`** function **must** return the actual power mode selected by the sensor driver.
- Req 3.3 The **`set_power_mode`** function **must** select an implemented power mode if that power mode is selected.
- Req 3.4 The **`set_power_mode`** function **must** keep the previous power mode **if** the requested power mode is not implemented or supported.
- Req 3.5 The **`set_power_mode`** function **must** implement `SensorPowerModePowerDown` and `SensorPowerModeSuspend`.
- Req 3.6 The **`set_power_mode`** function **may** select `SensorPowerModePowerDown` or `SensorPowerModeSuspend` if `SensorPowerModePowerDown` is requested.
- Req 3.7 The **`set_power_mode`** function **must** allow `SensorPowerModeLowPowerActive` **if** `SensorPowerModeActive` is requested for `BSX_INPUT_ID_MAGNETICFIELD`, `BSX_INPUT_ID_ACCELERATION`, and `BSX_INPUT_ID_ANGULARRATE`.
- Req 3.8 The **`set_power_mode`** function **may** select `SensorPowerModeLowPowerActive` or `SensorPowerModeActive` `if` `SensorPowerModeLowPowerActive` is requested and implemented.

- Req 3.9 The **`set_power_mode`** function **should** implement `SensorPowerModeInterruptMotion` for `BSX_INPUT_ID_ACCELERATION`.
- Req 3.10 The **`set_power_mode`** function **must** allow `SensorPowerModeOneShot` **if** `SensorPowerModeLowPowerActive` and `SensorPowerModeActive` are not allowed.
- Req 3.11 The sensor must transition from `SensorPowerModeOneShot` to `SensorPowerModeSuspend` or `SensorPowerModePowerDown` after the sample data was received when in `SensorPowerModeOneShot`.
- Req 3.12 The **`set_power_mode`** function **must** allow `SensorPowerModeLowPowerActive` and `SensorPowerModeActive` **if** `SensorPowerModeOneShot` is not allowed.
- Req 3.13 The **`set_power_mode`** function **must** allow `SensorPowerModeLowPowerActive` **if** `SensorPowerModeActive` is allowed.
- Req 3.14 The **`set_power_mode`** function **must not** modify the power mode of any other sensor driver.
- Req 3.15 A transition from `SensorPowerModePowerDown` to any other power state **must** complete within 1000 ms.
- Req 3.16 A transition from `SensorPowerModeSuspend` to any higher power state **must** complete within 100 ms.
- Req 3.17 The **`set_power_mode`** function **must** transition from `SensorPowerModeSelfTest` to `SensorPowerModeSuspend` or `SensorPowerModePowerDown` after a self-test has completed.
- Req 3.18 The **`set_power_mode`** function **must** call reportSelfTestResult after the sensor has transitioned from `SensorPowerModeSelfTest` to `SensorPowerModeSuspend` or `SensorPowerModePowerDown`.
- Req 3.19 The **`set_power_mode`** function **must** call sensorPowerModeChanged after the sensor has transitioned to the requested power.
- Req 3.20 The **`set_power_mode`** function **must** ensure that **`set_sample_rate`** is called if a sample rate transition is needed by the driver when transitioning from any active power mode to another active power mode.

- Req 4.1 The **`set_sample_rate`** function **must** be implemented if the sensor supports `SensorPowerModeLowPowerActive` or `SensorPowerModeActive`.
- Req 4.2 The **`set_sample_rate`** function **must** place the sensor in the maximum sample rate supported when a value of `0xFFFF` is specified.
- Req 4.3 The **`set_sample_rate`** function **must** place the sensor in the lowest non-zero sample rate supported when a value of 1 is specified.
- Req 4.4 The **`set_sample_rate`** function **must** return the actual sample rate selected by the sensor driver.
- Req 4.5 The **`set_sample_rate`** function **must** select the requested sample rate or higher, if possible.
- Req 4.6 The **`set_sample_rate`** function **must not** lower the sample rate of another sensor driver.
- Req 4.7 The **`set_sample_rate`** function **should not** increase the sample rate of another sensor driver unless the drivers are composites.
- Req 4.8 The **`set_sample_rate`** function **must** call `sensorRateChanged` after the desired sample rate has been set.
- Req 4.9 The **`set_sample_rate`** function **must** have the `maxRate` value set to the maximum rate supported by the sensor.
- Req 4.10 The **`set_sample_rate`** function **should** enforce rates as a power of two decimation from the max rate.
- Req 4.11 The **`set_sample_rate`** function **must** select a requested rate, if possible.

- Req 5.1 The **`set_dynamic_range`** function **must** be implemented for `BSX_INPUT_ID_ACCELERATION` and `BSX_INPUT_ID_ANGULARRATE`.
- Req 5.2 The **`set_dynamic_range`** function **should** support a range of +/- 16G for `BSX_INPUT_ID_ACCELERATION`.
- Req 5.3 The **`set_dynamic_range`** function **should** support a range of +/- 2000dps for `BSX_INPUT_ID_ANGULARRATE`.
- Req 5.4 The **`set_dynamic_range`** function **must** return the actual dynamic range selected by the driver.
- Req 5.5 The **`set_dynamic_range`** function **must** call `sensorRangeChanged` after the dynamic range has been set.

- Req 6.1 The **`get_dynamic_range`** function **must** be implemented for all sensor types.
- Req 6.2 The **`get_dynamic_range`** function **must** return the actual dynamic range for the sensors.

- Req 7.1 The **`enable_interrupts`** function **must** be implemented for all sensors.
- Req 7.2 The **`enable_interrupts`** function **must** be called before sensor data is transferred from the sensor.
- Req 7.3 The **`enable_interrupts`** function **must not** affect the interrupt status of another sensor.

- Req 8.1 The **`disable_interrupts`** function **must** be implemented for all sensors.
- Req 8.2 The **`disable_interrupts`** function **must** stop sensor data from being transferred from the sensor.
- Req 8.3 The **`disable_interrupts`** function **must not** affect the interrupt status of another sensor.
- Req 8.4 The **`int_enabled`** variable **must** be set if data is being transferred from the sensor.

- Req 9.1 The **`get_power_mode`** function **must** be implemented for all sensors.
- Req 9.2 The **`get_power_mode`** function **must** return the power mode of the driver.

- Req 10.1 The **`get_sample_rate`** function **must** be implemented if the sensor supports `SensorPowerModeLowPowerActive` or `SensorPowerModeActive`.
- Req 10.2 The **`get_sample_rate`** function **must** return the selected sample rate of the driver.

- Req 11.1 The **`get_scale_factor`** function **must** be implemented for `BSX_INPUT_ID_MAGNETICFIELD`, `BSX_INPUT_ID_ACCELERATION`, `BSX_INPUT_ID_ANGULARRATE`, `BSX_INPUT_ID_PRESSURE`, `BSX_INPUT_ID_HUMIDITY`, and `BSX_INPUT_ID_TEMPERATURE`.
- Req 11.2 The **`get_scale_factor`** function **must** return floating point scale factor to convert the sensor data into the correct units.

4.8 Example virtual sensor drivers

4.8.1 Continuous virtual sensor

The following example virtual driver is a simple continuous driver which consumes data directly from a physical driver.

```
////////////////////////////////////////////////////////////////////////
////////
///
/// @file  VirtPhysicalOutput.c
///
/// @project EM7189
///
/// @brief  Example driver used to report physical sensor data to the
host.
///
/// @classification Confidential
///
////////////////////////////////////////////////////////////////////////
////////
#include <SensorAPI.h>
#include <host.h>
#include "VirtPhysicalOutput.h"
#include <arc.h>
#include <FreeRTOS.h>

#define SENSOR_INPUT BSX_INPUT_ID_ACCELERATION  /* Physical sensor to
use for trigger source */
#define SENSOR_OUTPUT BSX_OUTPUT_ID_ACCELERATION_RAW /* Host output type
*/

typedef struct {
 SInt16  x;
 SInt16  y;
 SInt16  z;
} __attribute__ ((packed)) output_t;
```

Figure 15: Continuous Virtual Sensor – Header

```
static SensorStatus handle_sensor_data(VirtualSensorDescriptor* self,
void* data)
{
 output_t output;
 PhysicalSensorDescriptor* parent =
cast_HeaderToPhysical(getSensorParent(cast_VirtualToHeader(self)));

 float scaleAdjustment = self->expansionData.f32;
 float dynamicRange = getDynamicRange(cast_PhysicalToHeader(parent));
 // Scale to dynamic range, 16bit signed output
 float scaleFactor =
  parent->get_scale_factor(parent) * scaleAdjustment * (float)MAX_SINT16
/ dynamicRange;
 SystemTime_t timestamp = self->timestamp;
 SInt32* sendata = data;
 SInt32 xi, yi, zi;
 float x, y, z;

 portDISABLE_INTERRUPTS();

 xi = sendata[0];
 yi = sendata[1];
 zi = sendata[2];

 portENABLE_INTERRUPTS();

 x = xi * scaleFactor;
 y = yi * scaleFactor;
 z = zi * scaleFactor;

 x = SATURATE(MAX_SINT16, x, MIN_SINT16);
 y = SATURATE(MAX_SINT16, y, MIN_SINT16);
 z = SATURATE(MAX_SINT16, z, MIN_SINT16);

 output.x = (SInt16)x;
 output.y = (SInt16)y;
 output.z = (SInt16)z;

 reportSensorEvent(self, &output, timestamp);

 return SensorOK;
}
```

Figure 16: Continuous Virtual Sensor – Handle_Sensor_Data

```
VIRTUAL_SENSOR_DESCRIPTOR VirtualSensorDescriptor DESCRIPTOR_NAME = {
 .triggerSource = {
  .sensor = {
   .type = {
    .value = SENSOR_INPUT,
    .flags = DRIVER_TYPE_PHYSICAL_FLAG,
   },
  },
 },

 .physicalSource = {
  .sensor ={
   .type = {
    .value = SENSOR_INPUT,
    .flags = DRIVER_TYPE_PHYSICAL_FLAG,
   },
  },
 },

 .info = {
  .id = DRIVER_ID,
  .version = DRIVER_REV,
 },

 .type = {
  .value = SENSOR_TYPE_BSX(SENSOR_OUTPUT),
  .flags = DRIVER_TYPE_VIRTUAL_FLAG,
  .wakeup_ap = FALSE,
 },

 .expansionData = {
  .f32 = SCALE_FACTOR,
 },

 .priority = PRIORITY_2, // high priority

 .handle_sensor_data = handle_sensor_data,
 .outputPacketSize = sizeof(output_t),
};
```

Figure 17: Continuous Virtual Sensor – Virtual Sensor Descriptor

4.8.2 On-change virtual sensor

The following example shows the changes that are required to change a continuous virtual driver to an on-change driver. The key points for writing an on-change driver are to set the sensor descriptor to specify a type of `on_change` and ensure the `get_last_sensor_data` function is implemented. In the `handle_sensor_data` function, the data should only be sent if the data has changed. When reporting a sensor event in `get_last_sensor_data`, the timestamp of the previous data sample should be used.

```
// Implement the get_last_sensor_data function for an on-change driver
static SensorStatus virt_get_last_sensor_data(VirtualSensorDescriptor*
self)
{
 /* Use the (saved) timestamp from the previous sample sent in
      handle_sensor_data */
 reportSensorEvent(self, &output, timestamp);
 return SensorOK;
}

VIRTUAL_SENSOR_DESCRIPTOR VirtualSensorDescriptor DESCRIPTOR_NAME = {

 /* All other values are the same as the continuous virtual      sensor
*/

 .type = {
  .value = SENSOR_TYPE_BSX(SENSOR_OUTPUT),
  .flags = DRIVER_TYPE_VIRTUAL_FLAG,
  .wakeup_ap = TRUE, // Change required for on-change driver
  .on_change = TRUE, // Change required for on-change driver
 },

 .handle_sensor_data = handle_sensor_data,
 .get_last_sensor_data = virt_get_last_sensor_data, /* Change required
for on-change driver */
};
```

Figure 18: On-Change Virtual Sensor

4.8.3 One-shot virtual sensor

The following example shows the changes that are required to change an on-change virtual driver to a one-shot driver. The key points for writing a one-shot driver are to update the driver to disable itself after an event is thrown, and make sure the driver does not throw an event when turned on. These changes should be made in addition to the changes required for an on-change sensor driver.

```
static SensorStatus handle_sensor_data(VirtualSensorDescriptor* self,
void* data)
{
 output_t output;

 // Perform all required operations to calculate the output values

 reportSensorEvent(self, &output, timestamp);
 updateRequestedRate(cast_VirtualToHeader(self), 0.0F); // Change
required for one-shot driver

 return SensorOK;
}
```

Figure 19: One-Shot Virtual Sensor

4.9 Programming custom code extensions

4.9.1 Overview

On the BHy2 custom code extensions are implemented as hooks. The hook functionality in the BHy2 is a special mechanism which provides calling of multiple definitions of sensor interface hook functions at defined locations in the firmware according to priority level. When each main hook function is called by the firmware, all registered hook functions of that type are also executed. The supported hook types are summarized in Table 23 below.

The flow of hooks called during the start of execution is shown in Figure 20.

Table 23: Supported Hook Types

Hook type	Return type	Additional parameters	Notes
Procedures called during start and stop of execution			
`initOnce`	`void`	`None`	Called during start of execution, following hardware initialization. One time initialization code should be placed in this function.
`exitShutdown`	`void`	`None`	Called during start of execution, following host interface initialization. Initialization code should be placed in this function.
`initialize`	`void`	`None`	Called during start of execution, following host interface and sensor initialization. Initialization code should be placed in this function.
`teardown`	`void`	`None`	Called during stop of execution events following an initialization error. The BHy2 stops operation of all sensors. The user should ensure that custom code (such as a timer call-back) is disabled at this point.
Procedures tied to sensor interface			
`PhysicalRate`	`void`	`PhysicalSensorDescriptor* phys, float* rate`	Called any time a rate for a physical sensor driver is requested. The hook can overwrite the physical sensor's rate by writing a new value to the *rate parameter.

Hook type	Return type	Additional parameters	Notes
`TimerRate`	`void`	`VirtualSensorDescriptor* timer, float* rate`	Called any time a rate for a physical sensor driver is required. The hook can overwrite the sensor's timer rate by writing a new value to the *rate parameter.
`PhysicalRat e Changed`	`void`	`PhysicalSensorDescriptor* phys, float rate`	Called any time a rate for a physical sensor driver has changed. Used to notify listeners that a physical sensor driver rate has changed.
`PhysicalRan ge Changed`	`void`	`PhysicalSensorDescriptor* phys, UInt16 range`	Called any time the dynamic range for a physical sensor driver has changed. Used to notify listeners that physical sensor driver range has changed.
`VirtualSens ors Determined`	`void`	`None`	Called during checking of sensor state changes. Used to notify the listener that the framework has determined which sensors should be enabled. The hook can override or add additional requests here by setting the `info.status.enabled` bit in the desired sensor descriptor.
`OverrideMax Rate`	`void`	`SensorDescriptorHeader* sensor, float* rate`	Called during checking of sensor state changes. The hook can overwrite the maximal rate allowed for a given sensor by writing a new value to the *rate parameter.
`updatePhysi cal State`	`void`	`PhysicalSensorDescriptor* phys`	Called during sensor state changes. Notifies the user that the custom-controller sensor state changes should be done now.
`determinePo wer State`	`Sensor Power Mode`	`PhysicalSensorDescriptor* phys`	Called during sensor state changes. The default power mode can be overwritten by returning the desired power mode from the hook.

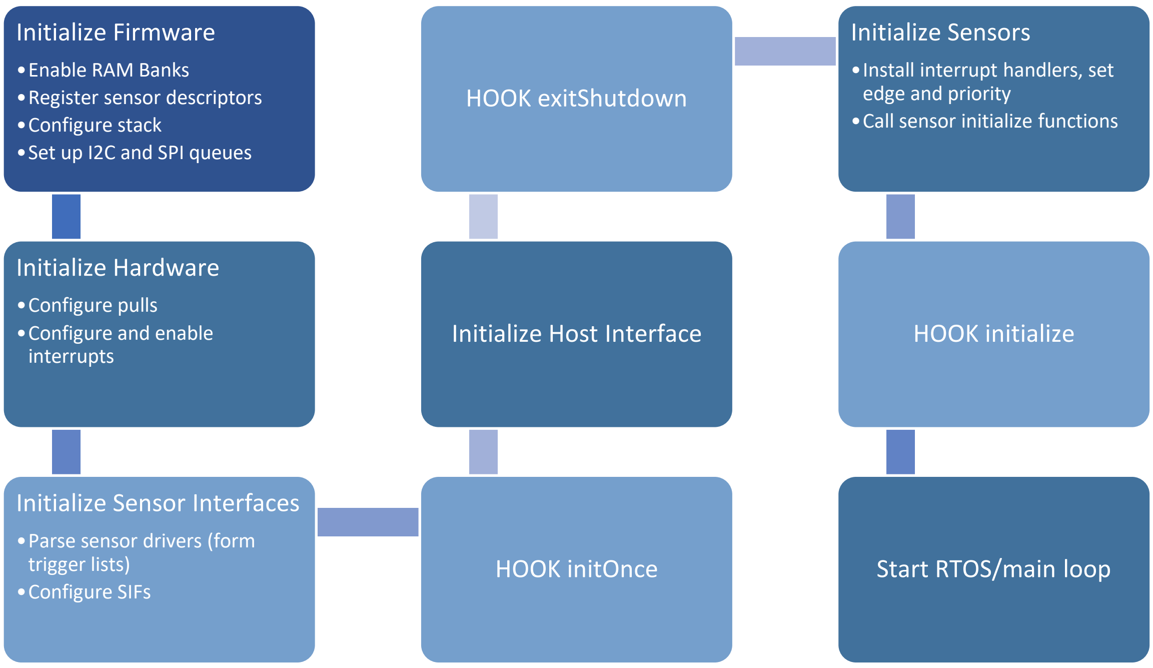

Initialize Firmware
• Enable RAM Banks
• Register sensor descriptors
• Configure stack
• Set up I2C and SPI queues
HOOK exitShutdown
Initialize Sensors
• Install interrupt handlers, set edge and priority
• Call sensor initialize functions
Initialize Hardware
• Configure pulls
• Configure and enable interrupts
Initialize Host Interface
HOOK initialize
Initialize Sensor Interfaces
• Parse sensor drivers (form trigger lists)
• Configure SIFs
HOOK initOnce
Start RTOS/main loop

Figure 20: Hooks Called During Initialization

4.9.2 Hook implementation

Sensor interface hook functions are defined and registered with the base firmware using the `HOOK` macro.

```
HOOK(__hook__, __function__, __priority__, __return__, ...)
{
   /* Function definition here */
}
```

Figure 21: Definition of Hook Function

The `HOOK` macro takes a number of arguments listed below:

- `__hook__` - The hook type to register
- `__function__` - The function to declare and register as a hook client
- `__priority__` - The priority of the hook, ranging from `HOOK_PRIORITY_MAX` (1) to `HOOK_PRIORITY_MIN` (10)
- `__return__` - The return type of the hook, specific to each hook type, see Table 23 Return Type column
- ... - Remaining arguments for the hook specific to each hook type, see Table 23 Additional Parameters column

New source files containing hook implementations can be added to the *$SDK/user/RamPatches* directory. Any new files in *the $SDK/user/RamPatches* directory should be added to the `RAM_PATCHES` variable in the CMake config file in the *$SDK/common* directory.

4.9.3 Hook priority level

Execution of hooks is ordered by priority level from `HOOK_PRIORITY_MAX` (1) to `HOOK_PRIORITY_MIN` (10).

```
// Hook priority definition
#define HOOK_PRIORITY_MAX  1
#define HOOK_PRIORITY_MIN  10
```

Figure 22: Hook Priorities

All kernel-level hooks of a given type will run to completion before any user-level hooks are executed. When two or more user-level hook functions with the same priority level are used, the order of their execution is unpredictable (it depends on internal sorting). For this reason, utilization of different priority levels is recommended.

4.9.4 Stopping hook execution

In some cases, execution of the rest of chained hook functions must be stopped to save execution time or due to a condition in the code. To achieve this, there is an extra parameter (`bool* stopHookExecution`) included by the `HOOK` macro for each hook function. Any defined hook function can set `stopHookExecution` to `TRUE` to stop execution of the rest of the registered hooks of the same type and lower priority. For detailed information, please, see section 4.8.6 that provides examples of hook usage.

4.9.5 Accessing data from hooks

Raw sensor data can be accessed by looking up the physical sensor using the `getPhysicalSensorDescriptorByType` or `getSensorSource` functions. Once the physical sensor descriptor has been located, the `descriptor->sensorData` variable can be used to access the raw sensor data. Note that interrupts must be disabled to make the reading of these data structures atomic.

4.9.6 Usage

This section contains two custom hook definition examples.

4.9.6.1 Hook example 1

The following hook example defines `myNewVSDHook` as a `VirtualSensorsDetermined` type hook. It has the lowest priority and will execute after all higher priority `VirtualSensorsDetermined` hooks.

```
#include <hooks_support.h>

HOOK(VirtualSensorsDetermined, myNewVSDHook, HOOK_PRIORITY_MIN, void)
{
  UNUSED(stopHookExecution);
  /* do something useful */
}
```

Figure 23: Hook Example 1

4.9.6.2 Hook example 2

The following hook example defines `myNewPRHook` as a `PhysicalRate` type hook. Because this hook sets `stopHookExecution` to `TRUE`, all other `PhysicalRate` user mode hooks with a priority lower than 5 will not be executed.

```
HOOK(PhysicalRate, myNewPRHook, 5, void,
   PhysicalSensorDescriptor* phys, float* rate)
{
  /* do something useful */

  // stop execution of the rest of chained hooks
  *stopHookExecution = TRUE;
}
```

Figure 24: Hook Example 2

4.10 Programming custom user mode libraries

Users can implement custom libraries into the SDK and call library functions within custom user image code, such as virtual sensors or hooks.

The following steps have to be executed to compile a library into user image:

1) Create a folder in *$SDK/libs*, containing:
   a) *<name>.c* source file(s) (names don't have to match the folder name, but need to be provided in *CMakeLists.txt*)
   b) *CMakeLists.txt* file, see example below (names are chosen arbitrarily)
   c) *includes/<name>.h* header file(s) exposing function prototypes and constants (names don't have to match the folder name, the directory including them has to be provided in *CMakeLists.txt*)
2) In */common/config.<dist_type>.cmake:*
   a) Add library name (folder name) to the `LIBRARIES` variable
   b) Add library name (folder name) to `BOARDS_LIBS` variable

```
get_filename_component( proj ${CMAKE_CURRENT_LIST_DIR}NAME)

project(${proj} C)

set(SOURCES
      customlib.c
)

include_directories(
      ../../libs/customlib/includes/
)

ADD_C_FLAGS(-DNO_JLI_CALLS)

ADD_ARC_LIBRARY(${proj}${SOURCES})

EXPORT_ARC_LIBRARY(${proj})
```

Figure 25: CMakeLists.txt Example

To use the library functions and data from inside custom user image code, the header files exposing these functions have to be included within the respective .c files. In order to avoid having to use the full path to the library from the root directory, users can add the path to the library to the `include_directories` variable of the *CMakeLists.txt* file of the calling component.

4.11 Using custom parameters

The primary control channel for configuring and querying the state of the system is done using parameter reads and writes. Current parameter use is summarized in Table 24. When sending a parameter request to the firmware, the upper byte is composed of a 0 (set) or 1 (get) in the upper nibble and the parameter page in the lower nibble. The second byte indicates the parameter number. More details on current parameters can be found in section 14 of the BHI260AB/BHA260AB Datasheet, Reference 1 and Reference 2.

Table 24: Parameters

Parameter Page	Parameter Command Range	Description
System (1)	0x0100 – 0x01FF	Meta Event Control, FIFO Control, Firmware Version, Timestamps, Framework Status, Virtual Sensors Present, Physical Sensors Present, Physical Sensor Information
Algorithm (2)	0x0200 – 0x02FF	Calibration state for physical sensors
Sensor Information (3)	0x0300 – 0x03FF	Sensor Information structure, including sensor type, driver ID, driver version, power, max range, min/max rate, and others. Read-only
Sensor Configuration (5)	0x0500 – 0x05FF	Sensor configuration parameters, including sample rate, maximum report latency, change sensitivity, and dynamic range. Read-only
Custom Parameter 1 (9)	0x0900 – 0x09FF	Available for custom use in the BHI260AB. May be used in other variants.
Custom Parameter 2 (10)	0x0A00 – 0x0AFF	Available for custom use in the BHI260AB. May be used in other variants.
Custom Parameter 3 (11)	0x0B00 – 0x0BFF	Available for custom use in the BHI260AB. May be used in other variants.
Custom Parameter 4 (12)	0x0C00 – 0x0CFF	Available for custom use
Sensor Control (14)	0x0E00 – 0x0EFF	Sensor specific control information, including FOC, OIS, and FST

This section describes how users can create their own custom parameters. Parameter pages 9 – 12 can be used to create custom parameters. Parameter numbers 1-255 are available for custom use within those pages. Parameter number 0 is reserved.

Note that parameter reads and writes should be used for infrequent control changes and infrequent data output. It is not recommended for high speed (> 1 Hz) sensor data output.

4.11.1 Initialization

Custom code must register the read and write callbacks for custom parameters by calling `registerReadParamHandler` and `registerWriteParamHandler` and passing in the handled parameter page and handler function. This can be done in an `initOnce` hook type. See Figure 26 for an example which registers new handlers for custom parameter page 9. After the parameter read and write handlers are registered, the firmware will route host requests for the custom parameters to the custom handlers appropriately.

```
#include <SensorAPI.h>
#define MY_PARAM_PAGE 9
extern bool myReadHandler(UInt8 param, UInt16 length, UInt8 buffer[],
             UInt16 *ret_length);
extern bool myWriteHandler(UInt8 param, UInt16 length, UInt8
buffer[]);

HOOK(initOnce, myInitOnceHook, HOOK_PRIORITY_RAM, void)
{
  registerReadParamHandler(MY_PARAM_PAGE, myReadHandler);
  registerWriteParamHandler(MY_PARAM_PAGE, myWriteHandler);
}
```

Figure 26: Parameter Page Read and Write Callback Registration

4.11.2 Parameter read handler

Custom code may implement a parameter read handler which copies the parameter data to the passed in buffer which is then returned to the host. As this function blocks much of the system, it is important that the data is copied to the buffer and returns quickly. Figure 27 includes an example for a parameter read handler.

```
bool myReadHandler(UInt8 param, UInt16 length, UInt8 buffer[], UInt16
*ret_length)
{
  // Save data from my code to host.
  switch(param)
  {
   case MY_PARAM:
     union {
      UInt8 *buffer;
      data_t *my_data;
     } conv;
     conv.buffer = buffer;
     // copy my_data fields into buffer here
     …
     *ret_length = sizeof(data_t);
     break;
   default:
     return FALSE; // unhandled parameter – indicate error to host
  }
  return TRUE;
}
```

Figure 27: Parameter Read Callback

4.11.3 Parameter write handler

Custom code may implement a parameter write handler which saves the host data into the stored parameter values. As this function blocks much of the system, it is important that the data is copied from the buffer and returns quickly. See Figure 28 for an example.

```
bool myWriteHandler(UInt8 param, UInt16 length, UInt8 buffer[])
{
  // Save data from host
  switch(param)
  {
   case MY_PARAM:
     // copy length bytes of data from buffer into my_data here
     …
     break;
   default:
   return FALSE; // unhandled parameter – indicate error to host
  }
  return TRUE;
}
```

Figure 28: Parameter Write Callback

4.12 Using general-purpose host registers

The BHy2 contains a fixed set of GPIO registers which can be used for communication between the firmware and the host.

In general, it is proposed to use the parameter interface or the sensor event concept for host communication, however, for some use cases a set of registers which is accessible both from the host and the firmware is sometimes easier to handle, since it carries a minimum overhead.

There is a register space of 12 bytes writeable by the host, and accessible read-only by the firmware, and another register space of 12 bytes writable by the firmware, and read-only accessible from the host.

API functions are available in `hif.h` for reading and writing these GPIO registers.

The HOST registers should be read using the `safeRead8`, `safeRead16`, or `safeRead32` API functions, passing in the address of the register to be read (e.g. `safeRead8(&HOST.Gp1.r8[0])`).

Table 25: Available GPIO Registers for Communication with Host

GPIO Register	I2C Register Address	Access Type	Host Access Type	Note
HOST.Gp1.r8[0]	0x08	RO	RW	General purpose input registers
HOST.Gp1.r8[1]	0x09	RO	RW	
HOST.Gp1.r8[2]	0x0A	RO	RW	
HOST.Gp1.r8[3]	0x0B	RO	RW	
HOST.Gp2.r8[0]	0x0C	RO	RW	
HOST.Gp2.r8[1]	0x0D	RO	RW	
HOST.Gp2.r8[2]	0x0E	RO	RW	
HOST.Gp2.r8[3]	0x0F	RO	RW	
HOST.Gp3.r8[0]	0x10	RO	RW	
HOST.Gp3.r8[1]	0x11	RO	RW	
HOST.Gp3.r8[2]	0x12	RO	RW	
HOST.Gp3.r8[3]	0x13	RO	RW	
PROC.Gp5.r8[0]	0x32	RW	RO	

GPIO Register	I2C Register Address	Access Type	Host Access Type	Note
PROC.Gp5.r8[1]	0x33	RW	RO	General purpose output registers
PROC.Gp5.r8[2]	0x34	RW	RO	
PROC.Gp5.r8[3]	0x35	RW	RO	
PROC.Gp6.r8[0]	0x36	RW	RO	
PROC.Gp6.r8[1]	0x37	RW	RO	
PROC.Gp6.r8[2]	0x38	RW	RO	
PROC.Gp6.r8[3]	0x39	RW	RO	
PROC.Gp7.r8[0]	0x3A	RW	RO	
PROC.Gp7.r8[1]	0x3B	RW	RO	
PROC.Gp7.r8[2]	0x3C	RW	RO	
PROC.Gp7.r8[3]	0x3D	RW	RO	

The access to these registers is performed asynchronously. The user has to take care that race conditions are avoided. E.g., when the firmware updates multiple output registers with a single 32-bit write while the host reads these register sequentially, some of the values read by the host may be updated, while others still have the old value.

4.13 Watchdog configuration

The watchdog timeout can be disabled, configured, enabled, and cleared using the APIs defined in *$SDK/common/7189/includes/watchdog.h*. The watchdog must be disabled before configuring the timeout. Example code to set the watchdog limit is shown in Figure 29.

```
// Set the watchdog limit to 10 ms
DisableWatchdog();
SetWatchdogLimit(getSYSOSCFrequency() * 10); // 10 ms
EnableWatchdogInterrupt();
```

Figure 29. Setting the Watchdog Limit

4.14 Firmware debugging

4.14.1 Debug message

For debugging a sensor driver the `fwrite`, `puts`, and `putchar` functions are available to store a message in the status FIFO buffer which can subsequently be read by the host. In addition the `printf` library is provided by stdio from MetaWare and can be enabled in order to have the capability to write an arbitrary string to the host.

Debug output is buffered in a 16-byte buffer. This buffer is sent to the host when full or when one of the debug functions output a linefeed. The host can cause a partial transfer to be sent by issuing a FIFO Flush command with the flush value set to `FLUSH ALL`.

As adding the `printf` library adds code space and all of these functions can dramatically affect timing, it is highly recommended that they are only used as a last resort and not included in production code.

4.14.2 Post mortem data

When a fatal error occurs due to either a processor exception, watchdog timeout, or an unrecoverable firmware error, the firmware saves the processor state of the BHy2 including the base registers, relevant auxiliary registers, and the stack. This debug data can be retrieved using the Download Post Mortem Data command. In response to this command the firmware will send the Crash Dump status block to the status FIFO.

See section 5 of the BHI260AB/BHA260 datasheets, Reference 1 and Reference 2, for more information on the download Post Mortem host command and Crash Dump Status Packet.

The `backtrace` tool in the SDK can be used to analyze the Crash Dump Status Packet.

4.14.2.1 Backtrace Utility

A backtrace utility is provided in the SDK to assist in decoding crash dump data. This utility is built during the normal build process and will be located in the *$SDK/build/bin* directory after completing a firmware build. After collecting the binary crash dump data, backtrace can be run to parse the data, as shown in Figure 30. *kernel_debug.elf* and *kernel-flash_debug.elf* are debug kernel elf files that include a small number of critical symbols from the ROM and kernel images to provide more information when decoding the crash dump data. These files are included in the SDK in the *$SDK/kernel* directory.

```
> backtrace pm.bin kernel_debug.elf [user.elf]
...
    r0 0x00128001
    r1 0x00a12e40
    r2 0x00136554
    r3 0x00a091c0
...
  gp r26 0x00a05c1c
  fp r27 0x00a11850
  sp r28 0x00a117f8
ilink r29 0x00124d5c (null)
   r30 0x30303030
blink r31 0x0013655c bmi160_accel_set_sample_rate_report_always
    pc 0x001029f2 NullHandler
   eret 0x001364c8 bmi160_accel_set_sample_rate
  erbta 0x001364b4 bmi160_accel_set_sample_rate
 erstatus 0x8000481e
   ecr 0x00020000
   efa 0x001364c8
  icause 0x00000000
 mpu_ecr 0x00000000

    diag 0x00000002
debug state 0x000000b2
 debug val 0x00000000
 error val 0x00000000
 interrupt 0x00000000
 err report 0x00000044

 stack start 0x00a05c1c
stack pointer 0x00a117f8
  stack size 0x00001000
 reset reason 0x00000004

  stack CRC 0xddd70c3f
     CRC 0x1aa74558

0x001029F2: NullHandler
    <r0>=0x00128001, <r1>=0x00a12e40, <r2>=0x00136554, <r3>=0x00a091c0,
    <r4>=0x00000000, <r5>=0x00a12e40, <r6>=0x00000000, <r7>=0x0000000f,
    <r8>=0x0000003f, <r9>=0x00a117bb, <r10>=0x10101010, <r11>=0x00000001,
    <r12>=0x00000001, <r13>=0x00a127f0, <r14>=0x00000004, <r15>=0x00a12ed4,
    <r16>=0x42c80000, <r17>=0x00a12e40, <r18>=0x00000008, <r19>=0x00000000,
    <r20>=0x00a12b40, <r21>=0x21212121, <r22>=0x22222222, <r23>=0x23232323,
    <r24>=0x24242424, <r25>=0x25252525, <r26>=0x00a05c1c, <r27>=0x00a11850,
    <sp>=0x00a117f8, <ilink>=0x00124d5c, <r30>=0x30303030, <blink>=0x0013655c
```

Figure 30 : Backtrace Utility

4.14.3 Current system time

The current system time can be determined by calling the `getSystemTime` function provided by the Timer library (prototype in *$SDK/libs/Time/includes/Timer.h)*.

4.14.4 Monitoring stack usage

The optional kernel firmware image RAM patch is available which reports stack space and usage. By default this RAM patch is included in the kernel firmware image.

After loading RAM/Flash firmware, stack information including task name, total stack size, and free/unused stack size is available by reading parameter ID 0x0110. See Figure 31 for an example.

```
   Stack Info
0x00: Task Name        : Idle
0x08: Total Stack Size     : 0x000007E8 (2024)
0x0C: Free Stack Size     : 0x00000694 (1684)
0x10: Task Name        : 4Virt
0x18: Total Stack Size     : 0x000009E8 (2536)
0x1C: Free Stack Size     : 0x0000091C (2332)
0x20: Task Name        : 3Virt
0x28: Total Stack Size     : 0x000009E8 (2536)
0x2C: Free Stack Size     : 0x000005F8 (1528)
0x30: Task Name        : 2Virt
0x38: Total Stack Size     : 0x000009E8 (2536)
0x3C: Free Stack Size     : 0x0000091C (2332)
0x40: Task Name        : Sensor
0x48: Total Stack Size     : 0x000009E8 (2536)
0x4C: Free Stack Size     : 0x00000710 (1808)
0x50: Task Name        : Host
0x58: Total Stack Size     : 0x00000BE8 (3048)
0x5C: Free Stack Size     : 0x00000A44 (2628)
0x60: Task Name        : CalibBSX
0x68: Total Stack Size     : 0x000017E8 (6120)
0x6C: Free Stack Size     : 0x00001570 (5488)
```

Figure 31 : Stack Usage Report

5 References

Reference 1: BHI260AB Datasheet (BST-BHI260AB-DS000)

Reference 2: BHA260AB Datasheet (BST-BHA260AB-DS000)

Reference 3: Synopsys MetaWare Website https://www.synopsys.com/dw/ipdir.php?ds=sw_metaware

Reference 4: Synopsys Github FOSS Toolchain for ARC® processors Website https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/releases

6 Legal disclaimer

6.1 Engineering samples

Engineering Samples are marked with an asterisk (*) or (e) or (E). Samples may vary from the valid technical specifications of the product series contained in this document. They are therefore not intended or fit for resale to third parties or for use in end products. Their sole purpose is internal client testing. The testing of an engineering sample may in no way replace the testing of a product series. Bosch Sensortec assumes no liability for the use of engineering samples. The Purchaser shall indemnify Bosch Sensortec from all claims arising from the use of engineering samples.

6.2 Product use

Bosch Sensortec products are developed for the consumer goods industry. They may only be used within the parameters of this product data sheet. They are not fit for use in life-sustaining or safety-critical systems. Safety-critical systems are those for which a malfunction is expected to lead to bodily harm, death or severe property damage. In addition, they shall not be used directly or indirectly for military purposes (including but not limited to nuclear, chemical or biological proliferation of weapons or development of missile technology), nuclear power, deep sea or space applications (including but not limited to satellite technology).

The resale and/or use of Bosch Sensortec products are at the purchaser's own risk and his own responsibility. The examination of fitness for the intended use is the sole responsibility of the purchaser.

The purchaser shall indemnify Bosch Sensortec from all third party claims arising from any product use not covered by the parameters of this product data sheet or not approved by Bosch Sensortec and reimburse Bosch Sensortec for all costs in connection with such claims.

The purchaser accepts the responsibility to monitor the market for the purchased products, particularly with regard to product safety, and to inform Bosch Sensortec without delay of all safety-critical incidents.

6.3 Application examples and hints

With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the device, Bosch Sensortec hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights or copyrights of any third party. The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. They are provided for illustrative purposes only and no evaluation regarding infringement of intellectual property rights or copyrights or regarding functionality, performance or error has been made.

7 Trademark notice

ARC® is a registered trademark of Synopsys Inc.

8 Document history and modifications

Rev. No	Chapter	Description of modification/changes	Date
1.4	All	Main release	2020-02-04
1.5	3, 4	Fixed typos and added missing descriptions	2020-05-28

Bosch Sensortec GmbH
Gerhard-Kindler-Straße 9
72770 Reutlingen / Germany

www.bosch-sensortec.com

Modifications reserved
Preliminary - specifications subject to change without notice
Document number: BST-BHI260AB-AN002-03
Revision: 1.5