Chipsmall Limited consists of a professional team with an average of over 10 year of expertise in the distribution of electronic components. Based in Hongkong, we have already established firm and mutual-benefit business relationships with customers from,Europe,America and south Asia,supplying obsolete and hard-to-find components to meet their specific needs.

With the principle of “Quality Parts,Customers Priority,Honest Operation,and Considerate Service”,our business mainly focus on the distribution of electronic components. Line cards we deal with include Microchip,ALPS,ROHM,Xilinx,Pulse,ON,Everlight and Freescale. Main products comprise IC,Modules,Potentiometer,IC Socket,Relay,Connector.Our parts cover such applications as commercial,industrial, and automotives areas.

We are looking forward to setting up business relationship with you and hope to provide you with the best service and solution. Let us make a better world for our industry!

Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832
Email & Skype: info@chipsmall.com Web: www.chipsmall.com
Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China

Atmel AVR4016: Sensors Xplained Software User Guide

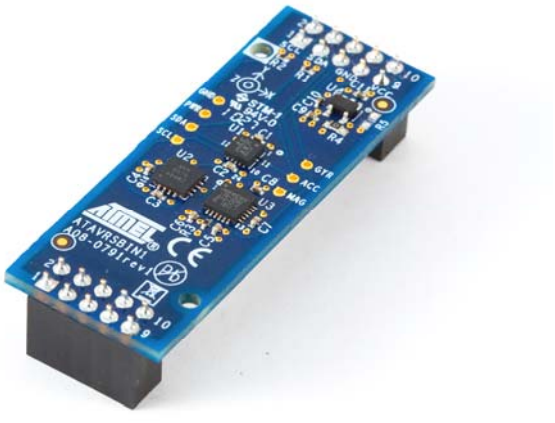
Features

- **Hardware-independent C language interfaces for sensor devices**
- **Conversion to standard units for all measurements**
- **Drivers for a variety of sensor types**
- **Easy-to-use configuration and initialization**

1 Introduction

This application note is an introduction to the Common Sensors Service in the Atmel® AVR® Software Framework (ASF). The Sensors Xplained software consists of a high-level, C/C++ application programming interface (API) and binary driver libraries for sensor devices on systems built around 8-bit and 32-bit Atmel AVR XMEGA® and Atmel AVR UC3 microcontrollers. ASF board support modules for the Atmel AVR Xplained MCU evaluation kits and Sensors Xplained add-on boards ("top modules") include configuration constants and runtime initialization calls that allow developers to pair AVR microcontrollers with different combinations of sensors on Sensors Xplained boards, and retarget standalone applications with little or no modification to the application source code. Demonstration projects included with the Sensors Xplained software illustrate how to bring together the sensor API, libraries, board support modules, ASF drivers, and configuration constants to build standalone applications.

Figure 1-1. Example Sensors Xplained add-on board.



Atmel Microcontrollers

Application Note

Rev. 8367B-AVR-06/11

2 Overview

The Atmel Sensors Xplained software is implemented as a common service extension to the Atmel AVR Software Framework (ASF), version 2.5 and later. The sensor software includes a high-level, portable C/C++ API, binary libraries containing sensor drivers, sensor configuration mechanisms, and example applications illustrating sensor API calls. Applications do not require any code that is specific to a particular sensor device. Instead, the application interacts with sensors in a device-independent manner and can be retargeted to different combinations of microcontrollers and sensors using a few basic configuration constants and linking against the appropriate driver library.

2.1 Common sensors service

The Sensors Xplained API portion is installed as a shared service in an ASF 2.x tree in the `common/services/sensors` directory, and includes API and driver header files, configuration header files for ASF services and target board support modules, and high-level utility functions.

Sensor device drivers and some interface routines in the Sensors Xplained API are distributed only as linkable binary modules, without source code. All applications written to this API must link against the appropriate library archive found in toolchain-specific subdirectories below the `thirdparty/sensors/libs` directory. In addition, the sensor hardware and platform interfaces make extensive use of conditionally compiled ASF services and platform interfaces. As a result, additional ASF source files are required when building an application that uses the Sensors Xplained libraries. These dependencies are managed by the Atmel AVR Studio® 5 project facility.

2.1.1 Sensors Xplained API modules

The `common/services/sensors` directory contains header files and C language implementation files defining the Sensors Xplained application programming interface (API). These include the `sensor.h` file (described below) and the `sensor_platform.c` source file that is part of the sensor configuration mechanism.

A sensor API hardware abstraction layer (HAL) acts as a translation layer between the AVR Software Framework drivers, sensor drivers, target board platform, and sensor API. These statically configured modules provide access to the various driver and board interfaces. The `sensor_platform_init()` routine is the primary runtime mechanism by which applications initialize the target Xplained evaluation board and Sensors Xplained board for use by the sensor API. Your application should call `sensor_platform_init()` as its first step during system initialization. It replaces the `board_init()` call that is typically made in other ASF-based applications.

The `common/services/sensors` directory also contains various other header and source files. These are used by modules within the sensor service. You should not include these files directly in your application or depend on the internal definitions, which are subject to change in future releases.

2.1.2 The sensor.h header file

The `common/services/sensors` directory contains the `sensor.h` header file, which is the primary file containing definitions that are required to use the Common Sensors Service. These definitions include function prototypes for all API interfaces, data

structure definitions, and various constants. The `sensor.h` file must be included in any application code that will use the sensor interfaces.

If you are using the Atmel AVR Studio 5 project tools to create your application, either from example applications or by selecting the sensor components to add to a custom application, the `avr.h` header file will be automatically modified to include `sensor.h`. In this case, your application does not need to include `sensor.h` directly – it may simply include `asf.h`, as usual, and the necessary definitions will be available.

NOTE

The code sequence examples that are found later in this document reference `sensor.h` explicitly. If you are using an automatically generated `asf.h` file that includes `sensor.h`, you may simply include `asf.h` instead.

2.1.3 The Sensors Xplained module_config directory

The C language header files located in `common/services/sensors/module_config` provide reference versions of the ASF configuration files that contain settings appropriate for use with the Sensors Xplained software.

If your application is based on the example sensor applications provided in AVR Studio 5, these configuration settings will automatically be included in your application build and will appear in the `config` directory within your project.

If you are adding sensor support to an existing or custom application using the AVR Studio 5 project facility, you may need to modify the initial settings in various configuration files (located in the `config` directory within your project) to match the settings found in the corresponding files in `common/services/sensors/module_config`.

2.1.4 Sensors Xplained drivers directory

The files located in the `common/services/sensors/drivers` directory supply definitions required by the Sensors Xplained hardware abstraction layer. Sensor applications do not require the definitions in these files, and you should not include these files or reference any of the symbols they define within your application. The definitions and API routines specified in the `sensor.h` file provide access to all installed sensor peripherals. None of the Atmel or third-party sensor driver implementations are provided in source code form, and all files within this directory are subject to change in future versions of the Common Sensors Service. The directory itself will be retained in the tree for those developers who are writing new sensor drivers for use by the sensor service.

2.1.5 Sensors Xplained driver libraries

Sensor drivers and API functions must be linked into your application from static link libraries built for the GCC and IAR™ Systems toolchains. The libraries are located in the `thirdparty/sensors/libs/gcc` and `thirdparty/sensors/libs/iar` directories, respectively. Only necessary modules will be linked into your final system image.

When you use the AVR Studio 5 project facility to create your application project, the correct library will automatically be included in your build.

NOTE

The sensor drivers and API functions found in the libraries are only available in binary format. No source code is provided.

The library name indicates the supported target AVR microcontroller, as well as whether or not the library is built with special flags targeting a build that will be used for debugging purposes.

The GCC driver libraries located in the `thirdparty/sensors/libs/gcc` directory have the following name formats:

```
libsensors-$mcu_series-debug.a
libsensors-$mcu_series-release.a
```

The IAR link libraries located in the `thirdparty/sensors/libs/iar` directory have a similar format:

```
libsensors-$mcu_series-debug.r82
libsensors-$mcu_series-release.r82
```

In both cases, *$mcu_series* identifies the specific 8-bit or/and 32-bit AVR microcontroller model being used. Table 2-1 lists the sensor driver libraries that are currently available.

Table 2-1. Sensors Xplained libraries.

Library name	Target MCU	Toolchain
libsensors-at32uc3a3-debug.a	AVR32 UC3-A3	GCC
libsensors-at32uc3a3-release.a	AVR32 UC3-A3	GCC
libsensors-at32uc3a3-debug.r82	AVR32 UC3-A3	IAR
libsensors-at32uc3a3-release.r82	AVR32 UC3-A3	IAR
libsensors-at32uc3a-debug.a	AVR32 UC3-A	GCC
libsensors-at32uc3a-release.a	AVR32 UC3-A	GCC
libsensors-at32uc3a-debug.r82	AVR32 UC3-A	IAR
libsensors-at32uc3a-release.r82	AVR32 UC3-A	IAR
libsensors-at32uc3b-debug.a	AVR32 UC3-B	GCC
libsensors-at32uc3b-release.a	AVR32 UC3-B	GCC
libsensors-at32uc3b-debug.r82	AVR32 UC3-B	IAR
libsensors-at32uc3b-release.r82	AVR32 UC3-B	IAR
libsensors-at32uc3c-debug.a	AVR32 UC3-C	GCC
libsensors-at32uc3c-release.a	AVR32 UC3-C	GCC
libsensors-at32uc3c-debug.r82	AVR32 UC3-C	IAR
libsensors-at32uc3c-release.r82	AVR32 UC3-C	IAR
libsensors-at32uc3l-debug.a	AVR32 UC3-L	GCC
libsensors-at32uc3l-release.a	AVR32 UC3-L	GCC
libsensors-at32uc3l-debug.r82	AVR32 UC3-L	IAR
libsensors-at32uc3l-release.r82	AVR32 UC3-L	IAR

2.2 Sensors Xplained target boards

In addition to the sensor service API header, source, and library files, all Atmel Sensors Xplained applications require target board support source files and board-specific configuration files. Board support files for AVR evaluation and development boards are located in the `avr32/boards` and `xmega/boards` subdirectories within the

ASF tree. The common board support files for the Atmel Sensors Xplained extension boards are located in the `common/boards/sensors_xplained` directory.

Rather than including the individual header files defined within each of these directories, applications and board interface software should simply include the `common/boards/board.h` file. This file is shared among all processor types, and exposes board-specific definitions based on the values of specific configuration constants, as discussed in following chapters.

3 Requirements

The following are the minimum requirements for creating standalone applications using the Sensors Xplained software.

- Atmel AVR Studio 5 (http://www.atmel.com/dyn/products/tools_card.asp?tool_id=17212)
- Atmel AVR Software Framework (ASF), version 2.5 or later, including header file updates for the 32-bit AVR toolchain
- Supported Atmel UC3 Xplained evaluation board
- Supported Atmel Sensors Xplained extension board
- Optional: IAR Embedded Workbench® for Atmel AVR32, version 3.31 or later (http://www.iar.com)
- Hardware programmer supported by the above tools (for example, Atmel AVR JTAGICE 3 or Atmel AVR One!)

AVR Studio 5 software is available on the Atmel website. Follow the accompanying installation instructions and application notes to install the development environment and toolchain.

The Common Sensors Service C/C++ source and header files, along with GCC and IAR Systems static link libraries containing sensor drivers and API functions, are included in the standard AVR Studio 5/ASF installation, but must be explicitly included in your application when you create and configure your project.

4 Creating an application

The Atmel AVR Studio 5/ASF installation contains several example applications, which illustrate how to use the Atmel Sensors Xplained API to control sensor devices and obtain measurement data. These applications, located in subdirectories below the ASF `common/applications/sensors` directory, illustrate how an application using the sensor API can be configured and built for various combinations of Xplained processor boards and Sensors Xplained add-on extension boards. New sensor API applications can be created by using the demonstration applications as templates, or by starting with a generic application and adding the sensor service and board support modules.

4.1 Example Sensors Xplained applications

Several example applications are included with the Sensors Xplained software to illustrate how the sensor interfaces are used. All of these applications may be found in the `common/applications/sensors` directory and use the same basic build mechanism and board definitions described earlier. Other example applications may be available in your particular installation.

- **Inertial Sensor Demonstration (inertial_demo)**
  This simple application obtains data from an inertial sensor board, including acceleration, rotation, magnetic heading, and temperature. The data are sent via a USB connection to a connected host PC for display using a terminal program.
- **Sensor Data Visualizer (inertial_visualizer)**
  This application also obtains sensor data from an inertial sensor board. The data are formatted into special packets and sent via a USB connection to a connected host PC for display using the special Atmel Data Visualizer application. See the AVR4017 – Atmel Data Visualizer application note, for more information.
- **Inertial Sensor Wakeup Demonstration (wake_demo)**
  This application demonstrates the use of the sensor event handing mechanism to wake up the system from a low-power sleep mode when a sensor event occurs. The event can be either a motion-threshold detection using an accelerometer or a new data event from a gyroscope.
- **Compass Sensor Calibration (compass_calibration)**
  This application demonstrates a basic, manual calibration sequence for compass/magnetometer devices.
- **Pressure Sensor Demonstration (pressure_demo)**
  This simple application obtains atmospheric pressure and temperature data from a pressure sensor board. The data are sent via a USB connection to a connected host PC for display using a terminal program.

4.1.1 Building an example application

The example sensor applications are selected and built in the same way as other ASF applications within AVR Studio. The following steps summarize how to create a new project based on the Inertial Sensor Demonstration project, which uses accelerometer, gyroscope, and compass devices.

1. In the AVR Studio 5 menus, select:
   File > New > New Project…
2. In the New Example Project window, select **Technology** in the left-hand panel.

3. Click on **Sensors** in the list of technology areas to display the sensor application choices.
4. Select the example project you wish to build. For this example, click on the Inertial Sensor Demonstration project for your Atmel Xplained processor board (for example, **Inertial Sensor Demonstration – UC3-A3 Xplained – AT32UC3A3256**). Click **OK**.
5. A Software License Agreement window will appear containing the license agreement for using the software contained in the Sensors Xplained library. If you agree to the terms of the license, click on the "I accept the license agreement" box and select **Finish**.
6. The example project files will appear within your Atmel AVR Studio windows. If the default sensors board for the project is the one you are using, you may now build and download the application normally. If you need to change the selection of the sensor board, see below.

NOTE

Most Atmel Sensors Xplained example applications, including the inertial sensor application summarized here, require a USB serial I/O connection to a virtual communication port on the host machine. Install the appropriate drivers on the host machine according to board setup instructions. The default application serial I/O configuration will transmit at 115,200 bits per second using 8-bit data, no parity, and one stop bit.

4.1.2 Changing the default sensor board

Each Sensors Xplained project has a default sensor board component defined. However, the sensor board selection can be changed based on the actual hardware you are using.

To display or change the current sensor board selection, use the following procedure within AVR Studio:

1. In the Solution Explorer pane, select the project name.
2. In the AVR Studio 5 menus, select:
   Project > Select Drivers from ASF…
3. The current sensor board selection will be in the Selected Modules list in the right-hand pane. (For example, for the Sensors Xplained Inertial 1 board, there will be an entry for "Sensors – ATAVRSBIN1 Sensor Board (Component).")
4. To change the sensor board:
   - Select the current sensor board component in the Selected Modules pane, and then click on **Remove from selection**
   - In the left-hand Available Modules pane, find and select the new sensor board you want to use, and then click on **Add to selection**
   - Click on **Finish**
5. You may now build and download your project normally.

4.2 Adding sensors to an existing application

The Common Sensors Service can be added to any application with only minor modifications by using the AVR Studio project facility. The steps in this section describe the overall procedure for a typical application that was originally created using the appropriate user application template for the processor board you are using.

First, you must add the sensor support modules into your application using the following sequence:

1. In the Solution Explorer pane, select the project name.
2. In the AVR Studio 5 menus, select:
   Project > Select Drivers from ASF…

3. To add the basic sensor service, find and select **Sensors – Sensor Device Stack** in the left-hand Available Modules pane, then click on **Add to selection**.
4. To add support for the sensor board you are using, find and select the new sensor board you want to use in the left-hand Available Modules pane, and then click on **Add to selection**.
   - For example, to add support for the Atmel Sensors Xplained Inertial 1 board, select **Sensors – ATAVRSBIN1 Sensor Board (Component)**
5. Click on **Finish**.

You must also modify your application to initialize the sensor service. To do so, make the following change in your application's `main.c` file:

Replace the following call:

```
board_init();
```

with:

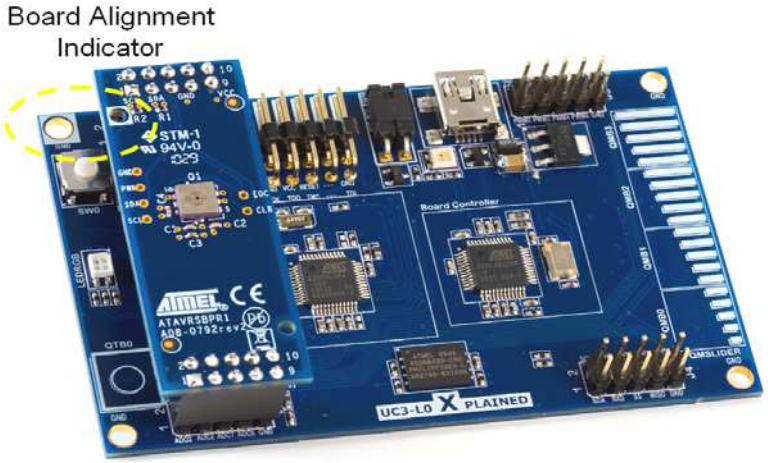
```
sensor_platform_init();
```

Once these modifications have been made, you can proceed to add the regular Sensors Xplained API function calls to your application. You will first need to initialize one or more sensor devices using `sensor_attach()`. Then you can perform control operations and read data using the interfaces and procedures described in Chapter 6, page 11, and Chapter 7, page 14.

4.3 Attaching a sensor board

Figure 4-1 shows how a Sensors Xplained sensor board attaches to an Xplained processor board. Note the white alignment indicators surrounding the mounting holes on both boards.

Figure 4-1. Attachment to UC3-L0 Xplained evaluation board.



5 Initialization

5.1 Configuration

As described above, the `common/services/sensors/module_config` directory contains various configuration files used by the example Atmel Sensors Xplained application projects. These are recommended settings for use with the Sensors Xplained functions, but they may be changed based on your application's requirements or other Atmel AVR Software Framework (ASF) dependencies.

When building an example application from Atmel AVR Studio 5 that uses the sensor interfaces, the appropriate configuration files will automatically be included in your application project and appear in your project's `config` directory.

If you are adding sensor support to an existing or custom application using the project facility, you may need to modify the initial settings in various configuration files (located in the `config` directory within your project) to match the settings found in the corresponding files in `common/services/sensors/module_config`.

5.2 Initializing sensors

The `sensor_attach()` function is used by your application to initialize a sensor and make it available for subsequent use. Your application simply specifies the type of sensor that is required and provides a descriptor structure that will be initialized. The sensor descriptor will then be used during later function calls to identify the sensor.

See the example code sequences in Chapter 7, page 14 for usage of the `sensor_attach()` function.

6 Control interfaces

The Atmel Sensors Xplained software supports both control and measurement operations for sensors in a device-independent manner whenever possible. This section describes various sensor control interfaces that allow your application to modify the behavior of the sensor device.

6.1 Sensor range

Sensor devices often provide multiple measurement ranges, which allow the device's available output resolution to be matched to the level of the physical conditions being measured. The sensitivity of the device is changed so that the full-scale output range of the device corresponds to different actual input level ranges. Therefore, the "raw" output value from the device for a given input level will change based on the range settings.

The Sensors Xplained sensor library functions automatically adjust their output scaling when the device's range is changed, and so the scaled numeric values that are returned to your application will be the same, subject to the limitations of the device resolution.

The `sensor_set_range()` function can be used to change the sensor range dynamically during execution of your application. The function takes the following form:

```
sensor_set_range (&device, range);
```

where `device` is the device descriptor of the device, and `range` is the range to be used. The `range` value is expressed in the same units used for normal, scaled output from the device (for example, milli-g for an accelerometer or Pascal for a pressure sensor). The `range` value is the number of measurement units from zero to full scale (positive or negative). For example, to set the range of an accelerometer device to cover -2000 to +2000 milli-g, the value of `range` would be 2000.

The value specified for `range` must match the settings that are valid for the device, or else an error is indicated (SENSOR_ERR_PARAMS).

See the individual driver descriptions in Chapter 9, page 27 for more information on valid range settings, default values, etc.

6.2 Sampling bandwidth

Sensor devices generally provide several different sampling frequencies or bandwidths. These different settings allow control over the tradeoff between measurement time and the stability (noise level) of the readings. Shorter measurement periods (higher sampling frequencies) reduce the time and power required to obtain each measurement, but the measured values will show higher variability, which appears as "noise" in the output values.

The `sensor_set_bandwidth()` function can be used to change the sensor sampling bandwidth dynamically during execution of your application. The function takes the following form:

```
sensor_set_bandwidth (&device, bandwidth);
```

where `device` is the device descriptor of the device, and `bandwidth` is the frequency to be used, in Hz.

The value specified for `bandwidth` must match the settings that are valid for the device, or else an error is indicated (SENSOR_ERR_PARAMS).

See the individual driver descriptions in Chapter 9, page 27 for more information on the valid bandwidth frequencies, default settings, etc.

6.3 Thresholds

Some sensor devices provide programmable thresholds that can be used to establish a measurement level for detection of certain events. A typical example is an accelerometer device which allows a motion detection threshold to be set – when the sensor detects motion exceeding the threshold value, an externally visible event is generated.

The `sensor_set_threshold()` function can be used to set these thresholds for devices which have such capabilities. The function takes the following form:

```
sensor_set_threshold (&device, type, value);
```

where `device` is the device descriptor of the device, `type` is the type of threshold being set, and `value` is the new threshold value. The value parameter is expressed in the same units normally used for reading scaled data from the sensor (for example, milli-g for an accelerometer device).

As an example, to set a motion detection threshold for an accelerometer device to 500 milli-g (0.5g), use the following:

```
sensor_set_threshold (&accel_dev, SENSOR_THRESHOLD_MOTION, 500);
```

Threshold detection is usually used together with sensor events. See Chapter 8, page 22 for more information on generating and using sensor events.

6.4 Calibration

Typically, sensors require some level of per-device calibration in order to provide accurate measurements. Often, the only required calibration is performed during the manufacturing of the device (factory calibration), and the calibration values are stored internally in the device. In other cases, it is necessary to calibrate the device in its actual deployed state. For example, compass/magnetometer devices typically are sensitive to the magnetic fields present in the final product (board, case, electrical connections), and these must be offset in order to obtain accurate readings.

When a sensor requires explicit calibration in its deployed environment, the resulting calibration values are often stored in nonvolatile memory within the microcontroller.

The `sensor_calibrate()` function allows your application to initiate and execute a calibration sequence for a sensor. The calibration sequence is specific to the sensor device. The `sensor_calibrate()` function takes a step number as an input parameter to support devices that require multi-step calibration sequences (for example, a series of measurements between which the user must physically manipulate the device).

See the individual driver descriptions in Chapter 9, page 27 for more information on calibration requirements and procedures.

6.5 Self test

Sensor devices often provide a self-test feature to provide a physical and/or electrical test of the sensor's operation. These tests are generally very device specific, as is the interpretation of the results.

The `sensor_selftest()` function provides a mechanism for invoking a sensor's self-test functions from your application. The function return value indicates the summary Pass or Fail result from the device test, along with a specific code indicating the failure type (if any).

The `sensor_selftest()` function also allows data values from the self test to be passed back to the caller in a generic manner – the specific returned data are specific to the device and its driver.

See the individual driver descriptions in Chapter 9, page 27 for more information on available self tests.

7 Reading sensor data

7.1 Overview

7.1.1 Sensor read interfaces

The Atmel Sensors Xplained software provides a set of high-level functions to obtain data from sensor devices and return the measurements in an easy-to-use form. Each type of sensor has a corresponding read function to get data from the device.

Table 7-1 contains a summary of the sensor read functions for each sensor type.

Table 7-1. Sensor read function summary.

Sensor type	Sensors Xplained function	Sensor_data_t field(s)	Measurement units
All	`sensor_device_id()`	`device.id` `device.version`	

Accelerometer (X, Y, Z)	`sensor_get_acceleration()`	`axis.x`
		`axis.y`
		`axis.z`
			Milli-g

Compass/Magnetometer	`sensor_get_heading()`	`heading.direction`
		`heading.inclination`
		`heading.strength`
			Direction: Degrees from magnetic north (0° to 360°)
			Inclination: Degrees from horizontal (-90° to +90°)
			Field strength: Microtesla (µT)
			(Note: One gauss (G) = 100µT)

	`sensor_get_field()`	`axis.x`
		`axis.y`
		`axis.z`
			Microtesla (µT)

Gyroscope (X, Y, Z)	`sensor_get_rotation()`	`axis.x`
		`axis.y`
		`axis.z`
			Degrees of rotation per second (°/s)

Pressure	`sensor_get_pressure()`	`pressure.value`	Pascal (Pa)
Temperature	`sensor_get_temperature()`	`temperature.value`	Degrees Celsius (°C)

NOTE

The code sequence examples found later in this section reference `sensor.h` explicitly. If you are using an automatically generated `asf.h` file that includes `sensor.h`, you may simply include `asf.h` instead.

7.1.2 Sensor data structure – sensor_data_t

All API functions that return sensor data readings do so using the `sensor_data_t` data structure. When the sensor read function returns, this structure will contain the measurement values from the device, as well as a high-granularity timestamp.

The `sensor_data_t` structure uses a C union to define “aliases” of the data fields to provide more meaningful names for use in your application. See Table 7-1 for the recommended field name references for specific functions.

The `sensor_data_t` structure also contains a special field which is set by your application to specify whether the sensor read function should return scaled units or raw readings. This field should be set before calling the sensor read function.

The final field in the `sensor_data_t` structure is a high-resolution timestamp value that provides an elapsed time value, expressed in microseconds (µs). This field is updated during each sensor reading using an internal Atmel AVR system clock.

7.1.3 Measurement units

The Atmel Sensors Xplained API functions provide sensor results in real-world (usually scientific or SI) units. These values are automatically scaled based on the current device settings. And so if the output range setting for a device is changed, for example, the scaled output will remain the same (subject to limitations of the device's precision in each range).

Many sensor readings are provided directly by the device, but require scaling or other conversion to SI units. Other results (for example, magnetic heading) are calculated by the Sensors Xplained functions based on lower-level sensor readings.

See Table 7-1 for the measurement units used for each type of sensor reading.

7.1.4 Reading "raw" values

Although it is normally preferable to obtain scaled values for sensor data, it is also possible to read the internal "raw" values from the sensor. Raw values may be useful for system setup or calibration, or for special operations that are specific to the sensor being used.

To read raw values from a sensor device, set the `scaled` field in the `sensor_data_t` data structure to `false` before calling the sensor's read function (for example, `sensor_get_acceleration()` or `sensor_get_pressure()`).

When raw values are returned, they are not modified by the read function, and so the actual values will vary, depending on the range setting for the device. However, calibration offsets (such as those used for compass/magnetometer sensors) will be applied to the returned values.

7.1.5 Timestamps

The `timestamp` field in the `sensor_data_t` structure is automatically filled with a microsecond (µs) value from the AVR controller's real-time clock when the sensor is read. These timestamps can be used to determine the relative timing of multiple sensor readings.

When the sensor read function returns, the timestamp can be read from the `sensor_data_t` structure's `timestamp` field.

7.2 Device ID and version

Most sensor devices provide an identifier value that can be read to determine the sensor model. In many cases, the device version can also be read. The `sensor_device_id()` function is a special routine that reads these values from the sensor device and returns them in a `sensor_data_t` structure, similar to the way that the actual sensor readings are returned. The function takes the following form:

```
sensor_device_id (&device, &id_data);
```

where `device` is the device descriptor of the sensor and `id_data` is a `sensor_data_t` structure to receive the ID and version data.

When the `sensor_device_id()` function returns, the device ID value can be read from the `sensor_data_t` structure's `device.id` field. The device version can be read from the `device.version` field. If either value is not provided by the sensor device, the corresponding field will be set to 0.

7.3 Acceleration

Accelerometer sensors measure linear acceleration force, typically along three axes (X, Y, and Z). The `sensor_get_acceleration()` function reads the sensor and returns the measured acceleration. The function takes the following form:

```
sensor_get_acceleration (&device, &accel_data);
```

where `device` is the device descriptor of the accelerometer and `accel_data` is a `sensor_data_t` structure to receive the acceleration data.

Scaled acceleration measurements are expressed in milli-g. When the function returns, the values can be read from the `accel_data` structure using the “axis” fields (`axis.x`, `axis.y`, `axis.z`).

7.3.1 Example code sequence

7.3.1.1 Definitions and declarations

```
#include "sensor.h"
sensor_t         accel_dev;    // device descriptor
sensor_data_t    accel_data;   // acceleration data from device
```

7.3.1.2 Sensor initialization

```
sensor_attach (&accel_dev, SENSOR_TYPE_ACCELEROMETER, 0, 0);
```

7.3.1.3 Sensor read

```
accel_data.scaled = true;      // read values in milli-g
sensor_get_acceleration (&accel_dev, &accel_data);
```

7.3.1.4 Use data in application

```
int32_t  app_x_value   = accel_data.axis.x;
int32_t  app_y_value   = accel_data.axis.y;
int32_t  app_z_value   = accel_data.axis.z;
uint32_t app_read_time = accel_data.timestamp;
```

7.4 Rotation

Gyroscope sensors measure rotation rates, typically along three axes (X, Y, and Z). The `sensor_get_rotation()` function reads the sensor and returns the measured rotation rate. The function takes the following form:

```
sensor_get_rotation (&device, &gyro_data);
```

where `device` is the device descriptor of the gyroscope and `gyro_data` is a `sensor_data_t` structure to receive the rotation data.

Scaled rotation rate measurements are expressed in degrees per second. When the function returns, the values can be read from the `gyro_data` structure using the “axis” fields (`axis.x, axis.y, axis.z`).

7.4.1 Example code sequence

7.4.1.1 Definitions and declarations

```
#include "sensor.h"
sensor_t       gyro_dev;       // device descriptor
sensor_data_t  gyro_data;      // rotation data from device
```

7.4.1.2 Sensor initialization

```
sensor_attach (&gyro_dev, SENSOR_TYPE_GYROSCOPE, 0, 0);
```

7.4.1.3 Sensor read

```
gyro_data.scaled = true;     // read values in degrees per second
sensor_get_rotation (&gyro_dev, &gyro_data);
```

7.4.1.4 Use data in application

```
int32_t  app_x_value   = gyro_data.axis.x;
int32_t  app_y_value   = gyro_data.axis.y;
int32_t  app_z_value   = gyro_data.axis.z;
uint32_t app_read_time = gyro_data.timestamp;
```

7.5 Compass heading

The most common use of a magnetic compass sensor is to obtain a direction heading relative to magnetic north for use in orientation and navigation applications. The Atmel Sensors Xplained API provides the `sensor_get_heading()` function to read the device and calculate such a heading value. The function takes the following form:

```
sensor_get_heading (&device, &compass_data);
```

where `device` is the device descriptor of the compass and `compass_data` is a `sensor_data_t` structure to receive the heading data.

The function returns three data values within the `sensor_data_t` structure using the "heading" fields:

The first value is the direction value, expressed in degrees (0 to 360, clockwise) from magnetic north. The direction value is the angle between the positive Y-axis of the device and the measured horizontal direction of the magnetic vector. The value can be read from the `compass_data.heading.direction` field.

The second value is the inclination angle expressed in degrees (-90 to +90) relative to horizontal. The inclination notation follows conventional usage in which positive values indication a downward (into the earth) angle and negative values indicate an upward angle. The inclination value can be read from the `compass_data.heading.inclination` field.

The third value is the net magnetic field strength (intensity) expressed in microteslas (µT). The value can be read from the `compass_data.heading.strength` field. This is a single net field strength value – to obtain separate readings of the field strength for

each directional axis (X, Y, and Z), use the `sensor_get_field()` function, discussed below.

NOTE

Magnetic compass sensors generally require a calibration procedure to correct for the local magnetic fields present in an actual device deployment. Without this calibration, the heading information will not be accurate. See the compass_calibration example application for an example of a manual calibration procedure.

7.5.1 Example code sequence

7.5.1.1 Definitions and declarations

```
#include "sensor.h"
sensor_t       compass_dev;    // device descriptor
sensor_data_t  compass_data;   // heading data from device
```

7.5.1.2 Sensor initialization

```
sensor_attach (&compass_dev, SENSOR_TYPE_COMPASS, 0, 0);
```

7.5.1.3 Sensor read

```
compass_data.scaled = true;    // read values in degrees and uTesla
sensor_get_field (&compass_dev, &compass_data);
```

7.5.1.4 Use data in application

```
int32_t  app_direction      = compass_data.heading.direction;
                                          // 0 to 360 deg
int32_t  app_inclination    = compass_data.heading.inclination;
                                          // -90 to +90 deg
int32_t  app_field_strength = compass_data.heading.strength;
                                          // uTesla
uint32_t app_read_time      = compass_data.timestamp
```

7.6 Magnetic field strength

In addition to reading a compass heading, a magnetic compass sensor can be used to obtain multi-axis measurements of the local magnetic field strength (intensity). The function takes the following form:

```
sensor_get_field (&device, &mag_data);
```

where `device` is the device descriptor of the compass and `mag_data` is a `sensor_data_t` structure to receive the magnetic field strength data.

Scaled magnetic field strength measurements are expressed in microteslas (µT). When the function returns, the values can be read from the `mag_data` structure using the “axis” fields (`axis.x, axis.y, axis.z`).

NOTE

Magnetic compass sensors generally require a calibration procedure to correct for the local magnetic fields present in an actual device deployment. Without this calibration, the field strength information will not accurately reflect the external environment of the sensor. See the compass_calibration example application for an example of a manual calibration procedure.

7.6.1 Example code sequence

7.6.1.1 Definitions and declarations

```
#include "sensor.h"
sensor_t       compass_dev;    // device descriptor
sensor_data_t  mag_data;       // magnetic data from device
```

7.6.1.2 Sensor initialization

```
sensor_attach (&compass_dev, SENSOR_TYPE_COMPASS, 0, 0);
```

7.6.1.3 Sensor read

```
mag_data.scaled = true;        // read values in uTesla
sensor_get_field (&compass_dev, &mag_data);
```

7.6.1.4 Use data in application

```
int32_t  app_x_value   = mag_data.axis.x;
int32_t  app_y_value   = mag_data.axis.y;
int32_t  app_z_value   = mag_data.axis.z;
uint32_t app_read_time = mag_data.timestamp
```

7.7 Atmospheric pressure

Atmospheric pressure is measured using a barometric pressure sensor. The `sensor_get_pressure()` function reads the sensor and returns the measured pressure. The function takes the following form:

```
sensor_get_pressure (&device, &press_data);
```

where `device` is the device descriptor of the pressure sensor and `press_data` is a `sensor_data_t` structure to receive the pressure data.

Scaled pressure measurements are expressed in Pascal. When the function returns, the value can be read from the `pressure_data` structure using the `pressure.value` field.

7.7.1 Example code sequence

7.7.1.1 Definitions and declarations

```
#include "sensor.h"
sensor_t       press_dev;      // device descriptor
sensor_data_t  press_data;     // pressure data from device
```

7.7.1.2 Sensor initialization

```
sensor_attach (&press_dev, SENSOR_TYPE_BAROMETER, 0, 0);
```

7.7.1.3 Sensor read

```
pressure_data.scaled = true;   // read values in pascals
sensor_get_pressure (&press_dev, &press_data);
```

7.7.1.4 Use data in application

```
int32_t  app_pressure  = press_data.pressure.value;
uint32_t app_read_time = press_data.timestamp;
```

7.8 Temperature

Unlike most other types of measurement, temperature readings may come from either a dedicated temperature sensor or from a device that has a different primary sensor function but can provide temperature data as a secondary output value.

Dedicated temperature sensors generally provide stable, high-accuracy readings. The secondary temperature data from other types of sensor devices are typically used internally for temperature compensation of the primary measurement, and these temperature readings often have fairly loose accuracy specifications.

The `sensor_get_temperature()` function allows access to temperature data from any sensors that support such measurements. No special device initialization is required to enable temperature measurement as a secondary function. The function takes the following form:

```
sensor_get_temperature (&device, &temp_data);
```

where `device` is the device descriptor of the sensor device to use for the temperature measurement and `temp_data` is a `sensor_data_t` structure to receive the temperature data.

Temperature data can be obtained from multiple sensor devices in a single application, if desired. Simply specify a different device descriptor when calling `sensor_get_temperature()`.

Scaled temperature measurements are expressed in degrees Celsius. When the function returns, the value can be read from the `temp_data` structure using the `temperature.value` field.

7.8.1 Example code sequence

This example shows how a temperature reading is obtained from both a dedicated temperature sensor and a gyroscope sensor. An equivalent sequence can be used to read the temperature from a different type of sensor that supports temperature measurement.

7.8.1.1 Definitions and declarations

```
#include "sensor.h"
sensor_t       temp_dev;       // device descriptor
sensor_data_t  temp_data;      // temperature data from device
```

7.8.1.2 Sensor initialization

To initialize a dedicated temperature sensor, use the following:

```
sensor_attach (&temp_dev, SENSOR_TYPE_TEMPERATURE, 0, 0);
```

Sensor initialization only needs to be done once per sensor device, even if it will be used for both temperature and another (primary) sensing function.

NOTE The specified type is for the primary function of the sensor (not the secondary temperature function).

For example, a single call to the following initialization function for a gyroscope device (which also supports temperature sensing) is all that is required to enable both rotation and temperature measurement:

```
sensor_attach (&gyro_dev, SENSOR_TYPE_GYROSCOPE, 0, 0);
```

The same descriptor (`gyro_dev`) would then be used during subsequent calls to either `sensor_get_rotation()` or `sensor_get_temperature()`.

7.8.1.3 Sensor read

```
temp_data.scaled = true;       // read values in degrees Celsius
sensor_get_temperature (&temp_dev, &temp_data); // temp sensor data
sensor_get_temperature (&gyro_dev, &temp_data); // gyro temp data
```

7.8.1.4 Use data in application

```
int32_t  app_temperature = temp_data.temperature.value;
uint32_t app_read_time   = temp_data.timestamp;
```

8 Handling sensor events

In addition to reporting measurements on demand, most sensor devices provide some mechanism to continuously monitor their physical surroundings and generate an externally visible event when certain criteria are met or internal conditions occur. The Atmel Sensors Xplained software provides support for handling these asynchronous events in a consistent manner across different sensor devices.

A sensor device announces an event by changing the level of a specific output pin. Each sensor device output pin is connected to an input pin on the Atmel AVR microcontroller. These connections are part of the hardware configuration information that the Sensors Xplained configuration automatically establishes for the specific sensor board and processor board combination. The AVR microcontroller input pin is then used to generate an interrupt when the level changes. The event support in the Sensors Xplained software allows your application to set up and enable an appropriate handler generically, without requiring specific references to the actual pin or interrupt source that is being used.

Once it has been set up and enabled, the event handler routine in your application will be called whenever the specified sensor event occurs. Your handler routine can then perform any appropriate action, such as changing the overall state of the application, processing new sensor data, providing an indication to the user, etc.

8.1 Adding an event handler

A sensor event handler is added by using the `sensor_add_event()` function. This function takes a single input parameter, the address of a `sensor_event_desc_t` event descriptor structure.

The `sensor_event_desc_t` structure contains various fields that specify the event handling behavior, including:

- the `sensor_t` descriptor of the sensor which will generate the event
- the type(s) of event to detect – to use a common handler for multiple events, specify the logical OR of multiple event types
- the address of the handler routine
- an argument to the handler routine (typically, the address of the `sensor_data_t` structure)
- whether sensor data from the event should be in scaled units or raw values
- whether the event is initially enabled or disabled

8.2 The event handler routine

The event handler routine is created as part of your application. This handler will be called when the corresponding sensor event occurs, after the sensor device driver has performed any necessary internal event servicing.

An event handler takes the following form:

```
void handler_name (volatile void * in);
```

The handler routine takes a single input parameter, a `void *`. This parameter is actually the address of the `sensor_event_desc_t` structure that was used when adding the event to the system. Within the `sensor_event_desc_t` structure, the `data`

field contains a `sensor_data_t` structure, which holds the actual sensor data obtained during the event.

An event handler can be defined to be called only when a single, specific event occurs, or a common handler can be defined which is called when any one of a set of events occurs. To determine the actual event which initiated the call to the handler, examine the event field within the `sensor_event_desc_t` structure, which will contain a `sensor_event_t` type code indicating the event.

NOTE

Your event handler routine will be called as part of the interrupt processing, asynchronously from the normal execution of your application. Because the handler executes at interrupt level, other interrupts will be masked (prevented from being serviced) while your handler is running. Therefore, you should structure your handler and application to require a minimum of processing in the handler itself, and perform subsequent work in your regular application code. This can be done by having the handler set flags or state variables, which are checked during the regular cyclic execution of your main program, to initiate additional actions to be taken.

8.3 Enabling and disabling events

After an event has been added, it can be dynamically enabled by calling the `sensor_enable_event()` function, or it can be disabled by calling the `sensor_disable_event()` function. The event handler routine remains defined even if the event is disabled, but the sensor device settings are changed so that the event interrupt will not be generated.

8.4 Events with no handler routine

Normally, you will define a handler routine to be called when a specified sensor event occurs so that it may process the sensor data or take other special action. However, it is possible to add and enable a sensor event without providing your own handler routine. This might be done when the simple act of the sensor generating an event interrupt provides the required effect. For example, if the microcontroller is placed into a low-power sleep mode, a motion detection interrupt from an accelerometer might be used to wake up the system, with no further processing required.

To set up event handling without providing your own handler, call the `sensor_add_event()` function as usual, but set the `handler` field to 0 (NULL).

8.5 Event types

There are many types of sensor events that might be generated, depending on the specific sensor device. See the device driver descriptions in Chapter 9, page 27 for more information on the events that can be generated by each individual sensor.

Possible event types include:

- SENSOR_EVENT_NEW_DATA – new sensor data are available
- SENSOR_EVENT_MOTION – device motion has been detected, often used with a programmable threshold for motion detection
- SENSOR_EVENT_LOW_G – low gravity (that is, free fall) detected
- SENSOR_EVENT_HIGH_G – high gravity (acceleration) detected
- SENSOR_EVENT_TAP – physical tap(s) on device detected
- SENSOR_EVENT_TILT – device tilt detected

8.6 Example code sequences

8.6.1 Motion detection event

The following example shows how to set up and use a motion detection event generated from an accelerometer sensor. The overall sequence will be similar for other sensor or event types.

8.6.1.1 Definitions and declarations

```
#include "sensor.h"
sensor_t                accel_dev;    // device descriptor
sensor_event_desc_t     accel_event;  // event descriptor
sensor_data_t           accel_data;   // data from event handler
```

8.6.1.2 Sensor and event initialization

```
sensor_attach (&accel_dev, SENSOR_TYPE_ACCELEROMETER, 0, 0);

accel_event = {
        .sensor      = &accel_dev,          // use accelerometer
        .event       = SENSOR_EVENT_MOTION, // motion detect event
        .data.scaled = true,                // return scaled data
        .handler     = accel_handler,       // address of handler
        .arg         = &accel_data,         // where to put data
        .enabled     = true                 // enable event
};

sensor_add_event (&accel_event);            // add event
```

8.6.1.3 Sensor event handler routine

```
void accel_handler (volatile void * in)
{
    // Set pointer to input descriptor address
    sensor_event_desc_t * const event = (sensor_event_desc_t *) in;

    // Copy data - note 'arg' = address of 'accel_data' structure
    *((sensor_data_t *)(event->arg)) = event->data;

    /* do other stuff (set a flag for application, etc.) */
}
```

8.6.2 Tap detection event

The following example shows how to set up and use a tap detection event generated from an accelerometer sensor. The initialization is somewhat different from other events because the tap detection involves a number of configurable timing and intensity threshold parameters. The `sensor_set_tap()` function is used to set the various parameters that affect tap detection.

Not all accelerometer devices can generate tap detection events. See the individual device driver descriptions in Chapter 9, page 27 to determine if SENSOR_EVENT_TAP is one of the supported event types.

8.6.2.1 Definitions and declarations

```
#include "sensor.h"
sensor_t                accel_dev;    // device descriptor
sensor_event_desc_t     tap_event;    // event descriptor
sensor_data_t           tap_data;     // data from event handler
sensor_tap_params_t     tap_params;   // tap detecion parameters
```

8.6.2.2 Sensor and event initialization

```
sensor_attach (&accel_dev, SENSOR_TYPE_ACCELEROMETER, 0, 0);

tap_params = {
        .count         = 2,           // detect up to 2 taps
        .axes = (SENSOR_TAP_AXIS_X | SENSOR_TAP_AXIS_Y |
                 SENSOR_TAP_AXIS_Z), // detect taps on all 3 axes
        .threshold_min = 0,           // use default min intensity
        .threshold_max = 0,           // use default max intensity
        .total_time    = 400,         // tap detect duration 400msec
        .tap_time_min  = 5,           // each tap must be >= 5msec
        .tap_time_max  = 50,          // each tap must be <= 50msec
        .between_time  = 300,         // gap between taps <= 300msec
        .ignore_time   = 100          // gap between taps >= 100msec
};

sensor_set_tap (&accel_dev, &tap_params);   // set tap parameters

tap_event = {
        .sensor      = &accel_dev,          // use accelerometer
        .event       = SENSOR_EVENT_TAP,    // tap detect event
        .data.scaled = true,                // return scaled data
        .handler     = tap_handler,         // address of handler
        .arg         = &tap_data,           // where to put data
        .enabled     = true                 // enable event
};

sensor_add_event (&accel_event);            // add event
```