AVR4016: Sensors Xplained Software Users Guide

AVR® Microcontrollers

Application Note

Features

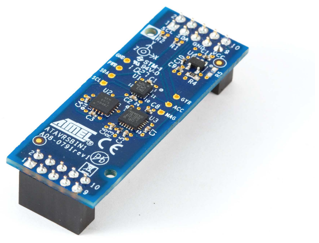
- **Hardware-independent C language interfaces for sensor devices**
- **Conversion to standard units for all measurement types**
- **Device drivers for a variety of MEMS-based sensors**
- **Easy-to-use configuration and initialization**

1 Introduction

NOTE: *This document describes the preliminary Beta release of the Atmel Sensors Xplained software including the core sensor management services. Additional features will be available in subsequent releases.*

This application note is an introduction to the Sensors Xplained 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- and 32-bit AVR (XMEGA^{TM} and UC3^{TM}) microcontrollers. ASF board support modules for the Atmel AVR Xplained 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 extension board



Rev. 4016A-AVR-01/11

2 Overview

The Sensors Xplained software is implemented as a common service extension to the AVR Software Framework version 2.2 or later, and consists of a high-level portable C/C++ API, binary libraries containing sensor and AVR platform drivers, application configuration and build rules, and demonstration applications illustrating sensor API calls and build mechanisms. Applications do not use sensor-specific code and can be retargeted to different combinations of evaluation and sensor boards using a few basic configuration constants and linking against the appropriate driver library.

2.1 Sensors Xplained 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, application Makefile configuration rules, configuration header files for ASF services and target board support modules, static link libraries for supported microcontrollers, and high-level utility functions.

Version 1.0 of the sensors service has the following directory structure:

```
C:\ASF_2\common\services\sensors
├───drivers
├───libs
├───math
├───module_config
├───physics
└───utils
```

All applications written to this API must link against the appropriate archive found in toolchain-specific subdirectories below the **common/services/sensors/libs** directory. In addition, the sensors hardware and platform interfaces make extensive use of conditionally compiled ASF services and platform interfaces. As a result, the **common/services/sensors/sensor_hal.c** source file and some additional ASF source files must be included in every high-level application build that uses the Sensors Xplained libraries.

For convenience, the **common/services/sensors/config.mk** build rules can be used by any sensor application built with GCC tools to set the correct toolchain include file paths and build the required ASF source files using a few basic build variables as described in the following sections. These Makefile rules can also be used as a reference list of the required header and source files for supported AVR microcontroller targets.

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). Your application must include the **sensor.h** file located from this directory to get the API constants, data structures, and function prototypes. In addition, the application must build the **sensor_hal.c** file and link the resulting object file into the application executable.

A sensor API hardware abstraction layer (HAL) defined in the `sensor_hal` modules 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 defined in the `sensor_hal` module is the primary runtime mechanism by which applications initialize the target Xplained evaluation board and Sensors Xplained board for use by the sensor API.

There are additional sensor HAL utilities that are potentially useful at the application level. However, some interfaces– such as the timer and delay routines – are simply wrappers used to isolate the sensor API implementation from architecture variations in the ASF driver interfaces, and these may change or be removed in future versions of the Sensors Xplained software. When there is a portable ASF driver or service interface, your application should use the general ASF interfaces instead of a functionally similar `sensor_hal` function.

2.1.2 Sensors Xplained Config Directory

The C language header files located in **common/services/sensors/module_config** contain tunable configuration constants that set operational parameters and configurable features for board support modules and common ASF services. Some of these modules do not have predefined values but are required when building the shared ASF services used in the Sensors Xplained API. If a required **conf_*xxx*.h** file does not modify tunable configuration constants, the ASF service uses default values and, in many cases, this is sufficient. All of the modules have default values that are suitable for use with supported Xplained evaluation boards without further editing, unless your application requires additional target board facilities, modifications to the clock source, etc. Of the existing configuration header files, the **conf_board.h** and **conf_clock.h** files are most frequently modified to customize target board behavior.

As noted in the document, AVR4007: AVR Software Framework 2 Getting Started, the board-specific configuration specified in **conf_board.h** defines various parameters for the target platform and is used to port an application between multiple boards. This file must be defined whenever any of the ASF target board support modules are built. The parameters and constants defined in this file vary from one target board to another. For an example, compare the definitions in the sensor service **conf_board.h** file with the contents of the following target board support modules used in the sensors demonstration application builds:

- *`avr32\boards\uc3_l0_xplained\uc3_l0_xplained.h`*
- *`avr32\boards\uc3_l0_xplained\init.c`*
- *`avr32\boards\uc3_a3_xplained\uc3_a3_xplained.h`*
- *`avr32\boards\uc3_a3_xplained\init.c`*

The **conf_clock.h** module defines clock sources and configurable parameters for the ASF **common\services\basic\clock** services. The sensor API relies upon the ASF **`sysclk`** API to initialize the platform system clock and all clocks derived from it. Available clock sources are specific to the target platform, and default values for a particular target board are defined in the board header files; see, for example, the **uc3_a3_xplained.h** file noted previously.

It is beyond the scope of this document to describe the **`sysclk`** API, microcontroller, and target board oscillator and clock configurations in detail. Documentation for the ASF common clock service, target board, and MCU will detail the available hardware facilities along with the high-level ASF interfaces that expose these to applications.

2.1.3 Sensors Xplained Drivers Directory

The header files located in the **common/services/sensors/drivers** directory supply definitions required in the current implementation of the Sensors Xplained hardware abstraction layer. Sensor service client 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 and third-party sensor implementations are currently available in source code form, and all existing header files within this directory are subject to change in future versions of the Sensors Xplained 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.

2.1.4 Sensors Xplained Driver Libraries

Sensor and AVR platform drivers must be linked into your application from static link libraries located in the **common/services/sensors/libs/gcc** and **common/services/sensors/libs/iar** directories, for GCC and IAR Systems toolchains, respectively. These libraries include all supported sensor drivers in addition to binary versions of certain ASF drivers, so that application builds do not need to rebuild these from source every time the sensor API is used. Only necessary modules are linked into the final system image. Note that the sensor drivers are only available in binary format, but the regular ASF drivers can be updated from sources located in the ASF tree.

The library name indicates the supported AVR microcontroller target, 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 **common\services\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 **common\services\sensors\libs\iar** directory have a similar format:

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

The *`$mcu_series`* identifies the specific 8-bit or and 32-bit AVR microcontroller model being used. Table **2-1** lists the available sensor driver libraries.

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 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 **avr32\boards** and **xmega\boards** subdirectories within the ASF tree. The common board support file for the Sensors Xplained extension boards is 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 include the **common/boards/board.h** file. This file is shared between all processor types and exposes board-specific definitions based on the values of specific configuration constants, as discussed in following sections.

Applications will generally need to add at least one board support file, **init.c**, from the appropriate board support directory to an application build, in addition to calling the *sensor_platform_init()* routine to initialize the target Xplained platform and sensor extension boards. Build configuration constants are used to specify the target board being used. Examples in following sections illustrate how static build configuration and runtime board initialization are accomplished for the UC3-A3 and UC3-L0 Xplained boards in the demonstration applications.

3 Downloading and Installing

At a minimum, developers will be required to install a GCC or IAR Systems toolchain and any supporting programming tools appropriate for the AVR or AVR32 microcontroller installed on the target Xplained evaluation board. In addition to the development toolchain and programmer, developers must install the Atmel AVR Software Framework (ASF) version 2.2 or better along with the Sensors Xplained service software and supporting static link libraries.

The following are the minimum requirements for creating standalone applications targeting AVR32 microcontrollers on an Xplained evaluation board:

- AVR32 Studio 2.6 or later, or IAR Embedded Workbench® for AVR32 3.31 and later (http://www.iar.com)
- Atmel Software Framework (ASF) 2.2 or later, including header file updates for the AVR32 toolchain
- Supported Xplained-series evaluation board
- Supported Sensors Xplained sensor extension board

In addition, a hardware programmer capable of interfacing AVR devices and supported by the above tools will be needed (e.g. JTAG/ICE MkII or AVR One).

***NOTE**: The Sensors Xplained software API is only compatible with ASF versions 2.2 and later. AVR UC3 Software Framework 1.7.0, included in AVR32 Studio 2.6, cannot be used to write applications to the API as described below. You must install the new ASF modules after the AVR32 Studio tools have been installed.*

Atmel AVR32 Studio software is available on the AVR32 Technical Library DVD, or on the Atmel website at http://www.atmel.com/products/avr32/ under the "Tools & Software" menu entry. Follow the accompanying installation instructions and application notes to install the development environment and toolchain.

Install the Atmel Software Framework by following links and installation instructions at http://asf.atmel.no/readme.html. If using AVR32 Studio 2.6, note that an ASF 2.2 installation will create an installation tree distinct from the AVR32 UC3 Software Framework 1.7 software and plug-ins integrated with the Eclipse-based IDE. The steps summarized in the following sections of this document will give a high-level overview of the ASF 2.x directory structure along with instructions on building applications written to the Sensors Xplained API within this framework.

The Sensors Xplained service C/C++ source and header files, along with GCC and IAR Systems static link libraries containing sensor and Xplained platform drivers, can be downloaded and installed by following the instructions and links at http://www.atmel.com/SensorsXplained/SensorSoftware.

After installing the basic minimal development tools, there are some additional installation details that should be addressed to ensure that software is updated and consistent. For AVR32 Studio versions built on the Eclipse development platform, make sure that plug-ins and tools are up to date by starting AVR32 Studio and selecting **Help > Check for Updates** in the main menu.

In addition to updating the installed Eclipse plug-ins, for the purpose of building and running applications using the ASF, ensure that the following additional installation items have been completed:

- Update toolchain header files using instructions and header files in the Atmel Software Framework

- Update system or command shell path settings with the locations of toolchain binaries and utilities

The ASF distributions include updated microcontroller header files that should be used to update the toolchain environment with new microcontroller definitions from the ASF releases. For example, the latest ASF version may be more recent than the latest available toolchain installation. As of ASF 2.2, the <***ASF_2*>/xmega/utils/header_files** directory contains header file updates and instructions for installing these files in a GCC or IAR Systems AVR toolchain installation, while the <***ASF_2*>/avr32/utils/header_files** directory contains header file updates and instructions for installing these files in a GCC or IAR Systems AVR32 toolchain installation, where <***ASF_2*>** is the root of the ASF installation tree.

In AVR32 Studio 2.6, the AVR32 toolchain binaries, header files, and utilities are installed as Eclipse plug-ins and, as a result, populated under the standard Eclipse plug-ins directory. Therefore, toolchain header and binary paths should be adjusted as described in the AVR32 Studio release notes documentation for this case.

AVR32 Studio 2.6 and later environments based on the Eclipse platform install toolchain binaries and header files, in addition to programmer utilities, as Eclipse plug-ins under the standard Microsoft® Windows® directory, **C:\Program Files\Atmel\AVR Tools\AVR32 Studio\plugins\.** Assuming ***$plugins*** is the root of the plug-ins directory, the toolchain header file updates must be populated in the directory:

$plugins\com.atmel.avr.toolchains.win32.x86_3.x.x.<build-id>\os\win32\x86\avr32\include\avr32

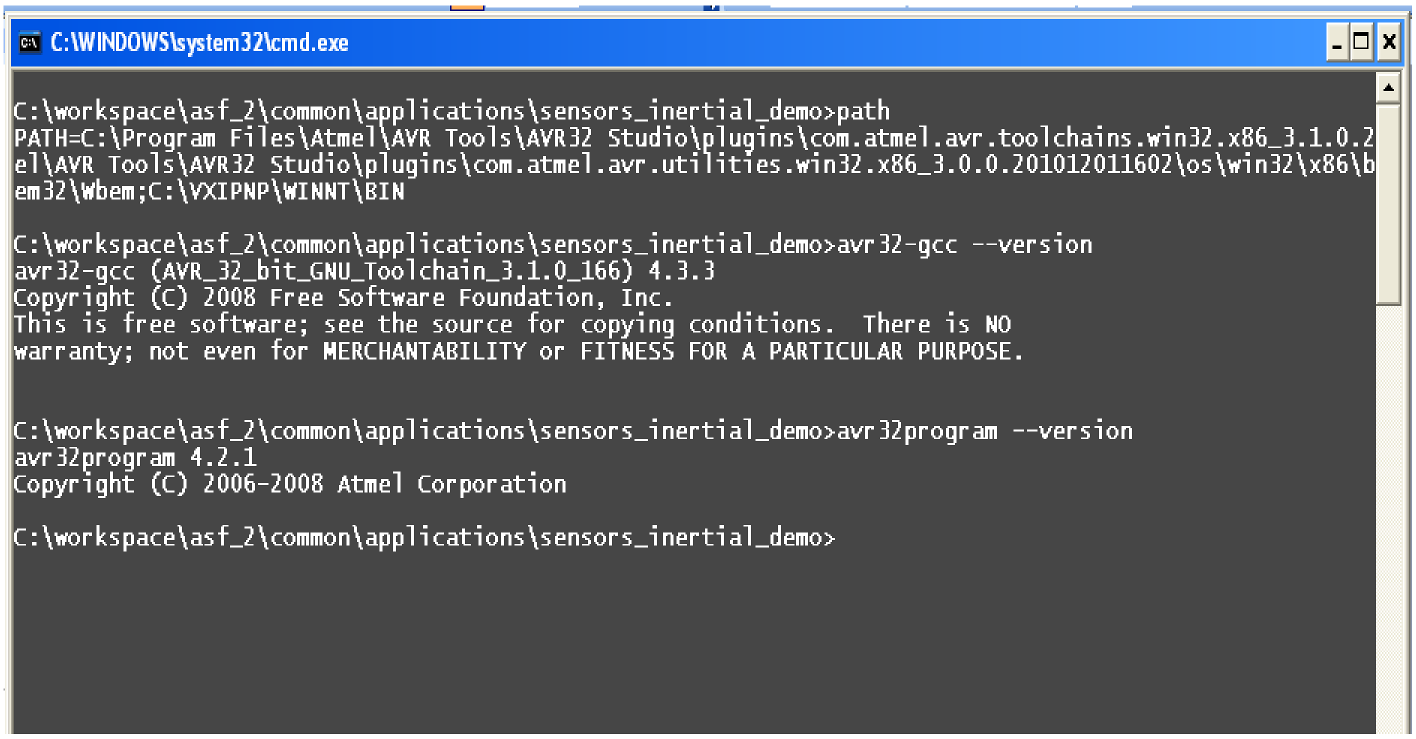
The plug-in version, "_3.x.x", will vary according to the availability and frequency of installation updates.

In order to provide command-line and project build access to the correct toolchain environment, the Windows system `PATH` variable should be updated to include this path along with the following directory:

$plugins\com.atmel.avr.utilities.win32.x86_3.x.x.<build-id>\os\win32\x86\bin

When new updates are installed via the AVR32 Studio menu option, these paths may be changed by updated toolchain and utility plug-in revisions. Making these paths available to a command-shell will facilitate building the demonstration application described below. Figure **3-1** illustrates a test of the basic command-line build environment for an AVR32 Studio 2.6 installation.

Figure 3-1: AVR32 Studio command-line environment



C:\WINDOWS\system32\cmd.exe

```
C:\workspace\asf_2\common\applications\sensors_inertial_demo>path
PATH=C:\Program Files\Atmel\AVR Tools\AVR32 Studio\plugins\com.atmel.avr.toolchains.win32.x86_3.1.0.2
el\AVR Tools\AVR32 Studio\plugins\com.atmel.avr.utilities.win32.x86_3.0.0.201012011602\os\win32\x86\b
em32\Wbem;C:\VXIPNP\WINNT\BIN

C:\workspace\asf_2\common\applications\sensors_inertial_demo>avr32-gcc --version
avr32-gcc (AVR_32_bit_GNU_Toolchain_3.1.0_166) 4.3.3
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.


C:\workspace\asf_2\common\applications\sensors_inertial_demo>avr32program --version
avr32program 4.2.1
Copyright (C) 2006-2008 Atmel Corporation

C:\workspace\asf_2\common\applications\sensors_inertial_demo>
```

4 Building An Application

Demonstration applications, located in subdirectories below the ASF **common/applications** directory, illustrate how an application using the sensor API can be configured and built for various combinations of Xplained evaluation boards and Xplained sensor add-on extension boards. Each application directory has **gcc/** and **iar/** subdirectories containing Makefiles for the GNU toolchain and EWAVR32 project files for the IAR toolchain, respectively. New sensor API applications can be created by using the demonstration build rules as templates. However, the required elements of an application build are relatively brief, and so it should be possible to create new projects – in AVR32 Studio, for example – with some basic knowledge of the required components.

4.1 Top-Level Application Makefile

All Sensors Xplained demonstration applications supply GNU Make definitions in **gcc/Makefile** and **gcc/config.mk** files. Application Makefiles use build targets (“make goals”) and build rules defined in one of the ASF utility Makefiles for XMEGA and AVR32:

- *xmega\utils\make\Makefile.in*
- *avr32\utils\make\Makefile.in*

So, for example, the Makefile implementation for each application simply contains a directive to include the appropriate ASF Makefile as follows:

```
#
#  Name:          Makefile
#  Contents:      Atmel Xplained Sensor Demo Makefile
#
include ../../../../avr32/utils/make/Makefile.in
```

4.2 Top-Level Configuration File

The top-level ASF **Makefile.in** files have a directive to include a user-defined **config.mk** file containing application-specific *make* variable definitions that configure and customize the predefined *make* rules for a particular project. *Make* variables that are defined or overridden in the application **config.mk** files are expressed in all upper-case characters, while variables that should not generally be modified by the *make* configuration files are defined using all lower-case characters. The Sensors Xplained **config.mk** files follow this convention when defining variables that should not be redefined by client applications.

Several user-defined *make* variables are optional, but a few will be required whenever the ASF **Makefile.in** files are used for an application build. In particular, the following variables must always be defined in the application **config.mk** file:

- `PRJ_PATH` - Specifies the top-level ASF directory relative to the application directory.
- `TARGET` - Specifies the build-target name and must be the same as the application directory name.
- `ARCH` - Specifies a GCC toolchain-specific machine architecture name (ucr2, ucr3, etc.).
- `PART` - Specifies a GCC toolchain-specific machine part name (uc3l064, uc3a3256, etc.).

Additional *make* variables are available to specify source files, compiler and linker flags, header file directory paths, etc. Consult the ASF documentation and ASF **Makefile.in** files more details on variables that configure the ASF top-level *make* rules.

As a convenience to developers, the Sensors Xplained service defines a set of common application *make* variable definitions in the **common/services/sensors/config.mk** file. All of the sensor demonstration application *make* rules for GCC use the definitions in this file in addition to application-specific variables defined in the application **config.mk** file. The Sensors Xplained common **config.mk** file is also a useful reference documenting the available *make* variables in addition to listing source files and header file paths that are required when building applications with the sensor API.

4.3 Sensors Xplained Application Example

Every Sensors Xplained API client application will require the following elements:

- Application code, target board support code modules, the **common/services/sensors/sensor_hal.c** module, and ASF common clock service source modules for the target AVR microcontroller must be compiled and linked into the application executable.
- Additional conditionally-compiled ASF common service modules supporting features such as formatted serial I/O, USB communications, etc. must be compiled and linked into the application executable.
- Toolchain and sensor API static link libraries must be linked into the application executable.

In addition, the project settings or *make* rules must also specify additional compiler and linker flags, header file and library paths, and possibly toolchain support modules such as C library startup files, for example.

As an example, observe how these elements are brought together to build the **common/applications/sensors_inertial_demo** for an Atmel Inertial One Sensor Board (ATAVRSBIN1) installed on either a UC3-A3 Xplained or UC3-L0 Xplained evaluation platform board.

The following steps assume the host development machine has all required development and programming tools, including the GNU Toolchain for AVR32, and ASF 2.2 or later development tree along with the Sensors Xplained software service and demonstration applications.

***NOTE**: Most Sensors Xplained demonstration 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. In case platform board USB drivers are not located, the Sensors Xplained software distribution package includes Microsoft® Windows® virtual port driver setup files for UC3-L0 Xplained and UC3-A3 Xplained kits in the* **`UC3L0-Xplained_AVR32_Virtual_Com_Port.inf`** *and* **`UC3A3-Xplained_AVR32_Virtual_Com_Port.inf`** *files, respectively.*

1. From a command shell, enter the **common/applications/sensors_inertial_demo/gcc** directory within the ASF tree.
2. Connect a UC3-L0 Xplained or UC3-A3 Xplained evaluation board to the host machine and programming tool. Ensure that an Inertial One sensor board is correctly installed on the Xplained board. The correct orientation and pin blocks can be determined by pairing the alignment indicators on the boards as demonstrated with the pressure sensor and UC3-L0 Xplained boards in figure **4-1**.
3. The **config.mk** configuration in this directory includes the Sensors Xplained top-level **config.mk** configuration values and defines the `PRJ_PATH`, `PART`, and `TARGET` variables in addition to application and target board source file names. When GNU `make` is invoked to build one of the targets defined in the top-level ASF **Makefile.in** file, users must provide variable definitions identifying the target platform and extension board as arguments. For example, build the application for a UC3-L0 Xplained board with an Inertial One add-on board as follows:

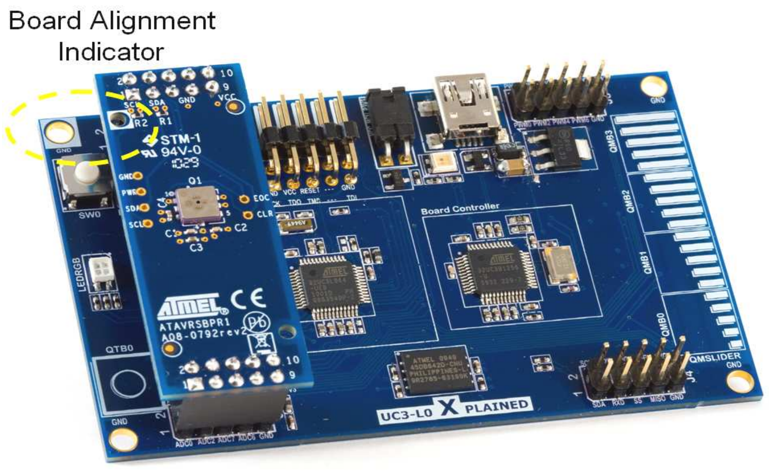
```
make board=UC3_L0_XPLAINED ext_board=SENSORS_XPLAINED_INERTIAL_1
```

4. Assuming a programmer is attached and powered, the resulting application can be programmed and run using the “program” and “run” build targets, respectively, as follows:

```
make board=UC3_L0_XPLAINED program run
```

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. The terminal application should be configured to append line feeds to incoming line ends for ASCII data.

Figure 4-1: Attachment to UC3-L0 Xplained Evaluation Board



The ASF **common/boards/board.h** file supplies target board definitions to applications based upon C-preprocessor configuration constants that must be defined when an application is built. The ASF **`BOARD`** and **`EXT_BOARD`** values must be set to one of the predefined configuration constants in the common **board.h** file. Application specific variables, **`board`** and **`ext_board`**, in the demonstration **config.mk** definitions are lower-case versions of the board configuration constants and should be assigned the same values. The Sensors Xplained application rules key other build variable definitions off of the specified target board constant.

For example, when `board=UC3_L0_XPLAINED` is provided as an argument to `make`, the `PART` make variable is set to "uc3l064" and a path to the **init.c** and **led.c** board support files is established, in addition to setting `BOARD=UC3_L0_XPLAINED` to bring in the correct board header file definitions from the common **board.h** file.

Within the top-level Sensors Xplained **config.mk** configuration, additional variables defining required ASF and library components are keyed off of the `PART` make variable defined in the application **config.mk** file.

All applications must link against one of the static Sensors Xplained libraries to include the code implementing the sensor API and drivers, in addition to select ASF driver modules that have been built and included in the archives.

***NOTE**: The ASF driver binaries included in the **libsensors-xxx** archives are provided as a convenience; developers are not required to add additional header file paths or driver source file names to every project build. There is nothing that has been customized within the ASF drivers for the sensor API. Developers may replace individual ASF driver modules or rebuild these from source as needed. However, the sensor drivers included in the **libsensors-xxx** archives are available only in binary form and therefore cannot be rebuilt from source modules in the ASF development tree.*

5 Initialization

5.1 Configuration

As described above, the **common/services/sensors/module_config** directory contains various configuration files that are used by the example 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 Software Framework (ASF) dependencies.

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 used during later function calls to identify the sensor.

See the example code sequences in Section 5 for usage of the *sensor_attach()* function.

6 Control Interfaces

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 such 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 setting.

The Sensors Xplained sensor library functions automatically adjust their output scaling when the device's range is changed, 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 form of:

```
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 Pascals for a pressure sensor).

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

See the individual driver descriptions in Section 7 for more information on the 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 form of:

```
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 an error is indicated (SENSOR_ERR_PARAMS).

See the individual driver descriptions in Section 7 for more information on the valid bandwidth frequencies, default settings, etc.

6.3 Calibration

Many sensors require some level of per-device calibration in order to provide accurate measurements. Sometimes, the only calibration is performed during the manufacturing of the device (i.e. factory calibration). 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.

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 who 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 Section 7 for more information on calibration requirements and procedures.

6.4 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 is specific to the device and its driver.

See the individual driver descriptions in Section 7 for more information on available self tests.

7 Reading Sensor Data

7.1 Overview

7.1.1 Sensor Read Interfaces

The 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 corresponding read function to get data from the device.

See Table 5-1 for a summary of the sensor read functions for each sensor type.

Table 5-1.

Sensor Type	Sensors Xplained Function	sensor_data_t Field(s)	Measurement Units

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

Sensor Type	Sensors Xplained Function	sensor_data_t Field(s)	Measurement Units

Compass	*sensor_get_heading()*	`field.heading`
		
		`field.inclination`
		
		`field.strength`
			Heading: Degrees from magnetic north (0° to 360°)
			Inclination: Degrees from horizontal (-90° to +90°)
			Field strength: uTesla
			(1 Gauss = 100 uTesla)

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

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

7.1.2 Sensor Data Structure – sensor_data_t

All API functions which 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 5-1 for recommended field names 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 (usec). This field is updated during each sensor reading, using an internal AVR system clock.

7.1.3 Measurement Units

The Sensors Xplained API functions provide sensor results in real-world scientific (SI) units. These values are automatically scaled, based on the current device settings. So, for example, if the output range setting for a device is changed, 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 (e.g. magnetic heading) are calculated by the Sensors Xplained functions based on lower-level sensor readings

See Table 5-1 for the measurement units used for each type of sensor.

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, calibration, or 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 (e.g. *sensor_get_acceleration()* or *sensor_get_pressure()*).

When raw values are returned, they are not modified by the read function. So, the actual values will differ depending on the range setting for the device.

7.1.5 Timestamps

The `timestamp` field in the sensor_data_t structure is automatically filled in with a micro-second 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 `timestamp` field.

7.2 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 form of:

```
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`).

7.2.1 Example Code Sequence

- Definitions and declarations

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

- Sensor initialization

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

- Sensor read

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

- Use data in application

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

7.3 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 form of:

```
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`).

7.3.1 Example Code Sequence

- Definitions and declarations

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

- Sensor initialization

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

- Sensor read

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

- Use data in application

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

7.4 Compass Heading

7.4.1 Example Code Sequence

- Definitions and declarations

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

- Sensor initialization

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

- Sensor read

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

- Use data in application

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

7.5 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 form of:

```
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 Pascals. When the function returns, the value can be read from the *pressure_data* structure using the `pressure.value` field.

7.5.1 Example Code Sequence

- Definitions and declarations

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

- Sensor initialization

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

- Sensor read

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

- Use data in application

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

7.6 Temperature

Many sensor devices can provide temperature data as a secondary output value. The temperature data is typically used internally in the device for temperature compensation, and these measurements sometimes have fairly loose accuracy specifications

The *sensor_get_temperature()* function allows consistent access to temperature data from any sensors that support such measurements. No special device initialization is required. The function takes the form of:

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

where *device* is the device descriptor of the sensor device to use for the temperature meaasurement, and *temp_data* is a sensor_data_t structure to receive the temperature data.

Temperature data can be obtained from multiple sensor devices, 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.

7.6.1 Example Code Sequence

This example shows how a temperature reading is obtained from a gyroscope sensor. An equivalent sequence can be used to read the temperature from a different sensor.

- Definitions and declarations

```
#include “sensor.h”
sensor_t       gyro_dev;      // device descriptor
sensor_data_t  temp_data;     // temperature data from device
```

- 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 that the specified type is for the primary function of the sensor (not the secondary temperature function).

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

- Sensor read

```
temp_data.scaled = true;     // read values in degrees Celsius
sensor_get_temperature (&gyro_dev, &temp_data); // NOTE gyro device
```

- Use data in application

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

8 Interrupt and Event Handling

The preliminary release of the Atmel Sensors Xplained software does not contain specialized support for interrupts and events that are triggered by sensor devices, but additional features are planned for the subsequent full release.

These functions will provide a set of high-level C interfaces for detecting and handling sensor events such as motion level thresholds and tap patterns. The interrupt support will facilitate use of low-power microcontroller modes with sensor-driven wakeup.

9 Sensor Device Drivers

The Sensors Xplained software is designed to provide a high-level set of interfaces that remain consistent across different sensor devices. However, different hardware devices ultimately do provide different features and settings, so some control interfaces are dependent on the specific device driver.

This section summarizes the capabilities that are driver-dependent for each supported device.

9.1 AKM AK8975 Compass / Magnetometer

9.1.1 Range

The AK8957 device does not provide different measurement ranges, so no modification is possible.

9.1.2 Sampling Frequency/Bandwidth

The AK8957 device does not provide different sampling frequencies, so no modification is possible.

9.1.3 Calibration

Like most compass/magnetometers, the AK8975 requires calibration to provide accurate measurements. The Sensors Xplained software includes a basic, manual calibration method that can be used to correct for constant magnetic offsets in your system (due to nearby metallic components, etc.). This calibration method requires that the device be repositioned between multiple steps of the calibration process.

See the **sensors_compass_calibrate** application for an example of using a three step manual calibration sequence. This application requires the user to alternately move the board to a specific position (laying flat, turned 180°, or inverted) and pressing a button on the processor board. After three such measurements, the three axes of the compass can be corrected.

9.1.4 Self Test

The AK8975 device provides a basic self-test which applies a known bias to the sensor device and confirms that the resulting sensor readings are within an expected range. Use the *sensor_selftest()* function with the SENSOR_TEST_DEFAULT type code. If the sensor is operating correctly, the function will return `true`. If the test fails, the function will return `false`.

9.2 Bosch BMA150 Accelerometer

9.2.1 Range

The BMA150 device provides the following range settings, expressed in milli-g:

- 2000 (+/- 2g)
- 4000 (+/- 4g)
- 8000 (+/- 8g)

The default setting is 4000 (+/- 4g).

9.2.2 Sampling Frequency/Bandwidth

The BMA150 device provides the following sampling frequency settings, expressed in Hz:

25, 50, 100, 190, 375, 750, 1500

The default setting is 1500 Hz.

9.2.3 Calibration

The BMA150 device does not require calibration.

9.2.4 Self Test

The BMA150 device provides a basic self-test which applies a known bias to the sensor device and confirms that the resulting sensor readings are within an expected range. Use the *sensor_selftest()* function with the SENSOR_TEST_DEFAULT type code. If the sensor is operating correctly, the function will return `true`. If the test fails, the function will return `false`.

9.3 Bosch BMP085 Pressure Sensor

9.3.1 Range

The BMP085 device does not provide different measurement ranges, so no modification is possible.

9.3.2 Sampling Frequency/Bandwidth

The BMP085 device does not provide different sampling frequencies, so no modification is possible.

9.3.3 Calibration

The BMP085 device does not require calibration.

9.3.4 Self Test

The BMP085 device does not provide a self-test function.

9.4 Invensense ITG-3200 Gyroscope

9.4.1 Range

The ITG-3200 device provides only one operating range: +/- 2000 degrees per second.

9.4.2 Sampling Frequency/Bandwidth

The ITG-3200 device provides the following sampling frequency settings, expressed in Hz:

5, 10, 20, 42, 98, 188, 256, 2100

The default setting is 256 Hz.

9.4.3 Calibration

The ITG-3200 device does not require calibration.

9.4.4 Self Test

The ITG-3200 device does not provide a self-test function.

10 Example Applications

Several example applications are included with the Sensors Xplained software, to illustrate how to use the sensor interfaces. All of these applications may be found in the **common/applications** directory and use the same basic build mechanism and board definitions described earlier.

- **sensors_inertial_demo** – A simple application which obtains data from an inertial sensor board, including acceleration, rotation, magnetic heading, and temperature. The data is sent via a USB connection to a connected host PC for display using a terminal program.
- **sensors_inertial_visualizer** – This application also obtains sensor data from an inertial sensor board. The data is formatted into special packets and is sent via a USB connection to a connected host PC for display using the special Atmel Data Visualizer application. See document AVR4017 – Atmel Data Visualizer application for more information.
- **sensors_compass_calibrate** – This application demonstrates a basic, manual calibration sequence for compass/magnetometer devices.
- **sensors_pressure_demo** – A simple application which obtains atmospheric pressure and temperature data from pressure sensor board. The data is sent via a USB connection to a connected host PC for display using a terminal program.
- **sensors_altitude_demo** – An example application which obtains atmospheric pressure and temperature data from pressure sensor board and uses this to calculate altitude above sea level. The data is sent via a USB connection to a connected host PC for display using a terminal program.

11 Document Revision History

Table 9-1 – Document Revisions

Revision	Date	Summary
A	01/2011	Initial version for preliminary / Beta release.

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Milennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chou-ku, Tokyo 104-0033
JAPAN
Tel: (+81) 3523-3551
Fax: (+81) 3523-7581