InvenSense Inc.
1197 Borregas Ave., Sunnyvale, CA 94089 U.S.A.
Tel: +1 (408) 988-7339 Fax: +1 (408) 988-8104
Website: www.invensense.com

Date: 8/4/2011

Atmel UC3 Reference Implementation User Guide for InvenSense Embedded MotionApps™ Platform Release 2.0.0

A printed copy of this document is
NOT UNDER REVISION CONTROL
unless it is dated and stamped in red ink as,
"REVISION CONTROLLED COPY."

Table of Contents

1 Revision History

Date	Rev	Description
07/22/2011	1.0	Initial Release
08/03/2011	1.1	Additions regarding 6-axis only Embedded MotionApps
08/04/2011	1.2	Corrections and edits after internal review

2 Purpose and Scope

This document describes the InvenSense Embedded MotionApps Platform System Layer implementation. It is intended as a user guide for programmers who want to integrate Embedded MotionApps Platform onto an embedded platform. An implementation of Embedded MotionApps Platform on the Atmel UC3 Xplained development platform is presented as an example. This document is intended to be used alongside the Embedded MotionApps Platform Reference Manual, a comprehensive description of the Embedded MotionApps Platform APIs. Also, this document may refer to terms Embedded MotionApps Platform or Embedded MPL interchangeably.

3 Embedded MotionApps Platform Overview

To aid developers to rapidly develop and deploy embedded application using IMU/MPU, InvenSense provides Embedded MotionApps Platform, which can be easily integrated into an existing system. Embedded MotionApps Platform contains algorithms to perform Motion Processing with sensor fusion and provides APIs to easily access the resulting sensor data.

The different layers of Embedded MotionApps Platform are as shown in Figure 1.

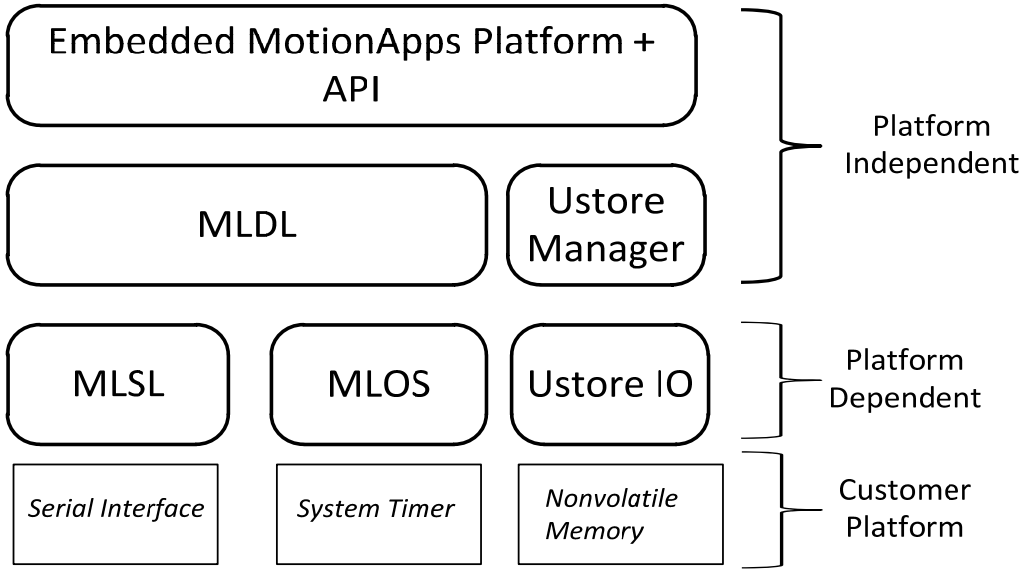

Figure 1: Embedded MotionApps Platform

The core of the Embedded MotionApps Platform is platform independent.

- The MotionApps Platform APIs provide access to sensor data, and allow the user to select the power mode for the sensor devices.
- The MLDL is the driver layer which manages the interface with the MPU device.
- UStore Manager handles loading and storing of calibration data.

The Embedded MotionApps platform interfaces with the customer platform through the following modules:

- MLSL is the platform dependent implementation for serial communication. The Embedded MotionApps platform uses this to communicate with the MPU device over an I^2C bus.
- MLOS provides an interface to a system timer and delay routine. The Embedded MotionApps platform requires a system timer with millisecond resolution.
- UStore IO provides an interface to nonvolatile memory, This may be a filesystem, on-board EEPROM, external serial Flash device, etc.

3.1 **Embedded MotionApps Platform Power Modes**

The Embedded MotionApps Platform has a power mode API which allows the user to select any power mode.

`umplInit` must be called before any other MotionApps Platform APIs can be used. After `umplInit` is called, all devices will still be powered off. The user can then call any of the four power control functions below to select a power mode.

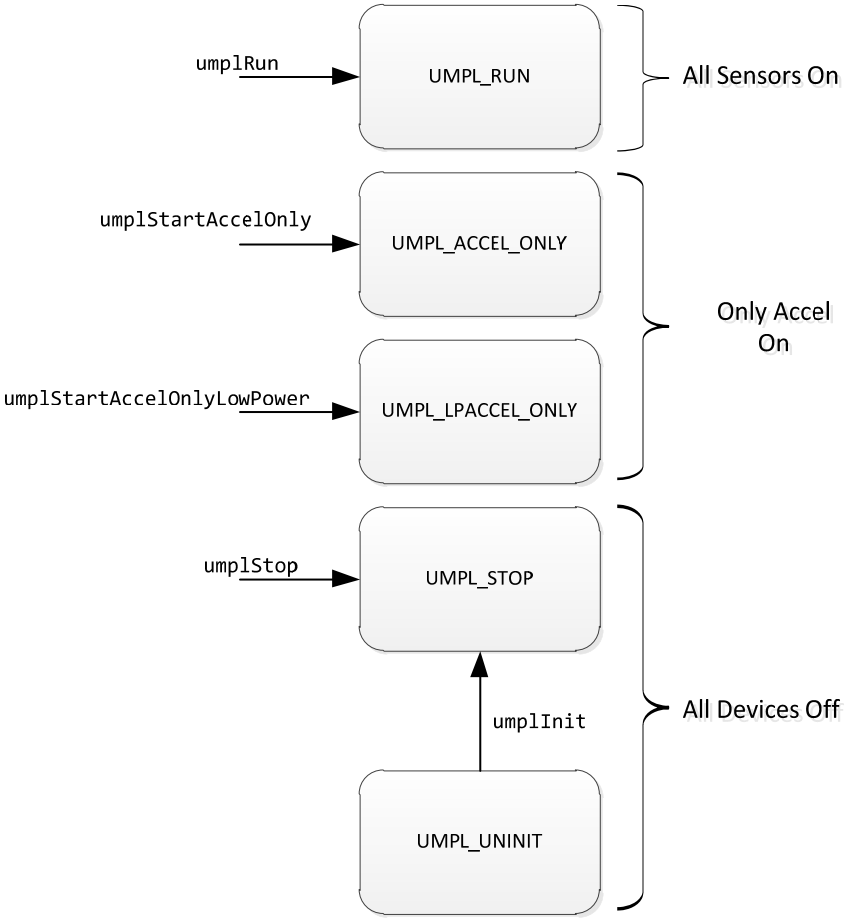

Figure 2: Embedded MotionApps Platform State Machine Diagram

Note:

- When built for 6-axis, UMPL_RUN will power on the accelerometer and gyros. When built for 9-axis, UMPL_RUN will power on the accelerometer, gyros, and magnetometer.
- Accelerometer only power modes are only supported with the MPU-6050, revision B.

4 Embedded MotionApps Platform Integration

An AVR studio 5 solution is provided as an example for Embedded MotionApps platform integration. Extract the Embedded MotionAps release package and copy the "umpl" folder located in src\targets\uc3\bsp\ into a suitable location in your project. Implement the platform dependent modules, i.e., MLSL, MLOS and Ustore IO, as explained in the sections below. Add the source path for the "umpl" folder and the right compiler flags to your project property. All the supported compiler flags are described in section 4.12. Depending on the motion sensors being used, different .c files need to be compiled. This is explained in section 4.9. Finally, platform specific data structures need to be initialized. This is further described in section 4.10.

4.1 Motion Library System Layer (MLSL)

The MLSL requires the user to implement each of these functions for their platform. A reference MLSL is provided in **platform/at32/mlsl_at32.c**. The serial functions use the twim library provided in Atmel AVR Studio 5 for the Atmel AVR32 microcontroller.

4.2 MLSL Error Codes

Each function in the MLSL returns an `inv_error_t`, which is defined, along with INV_ERROR macros, in **platform/include/mltypes.h**.

4.3 Serial Open, Reset, and Close

- `inv_error_t inv_serial_open(char const *port, void **sl_handle)`
- `inv_error_t Inv_serial_reset(void *sl_handle)`
- `inv_error_t Inv_serial_close(void *sl_handle)`

The Embedded MotionApps Platform MLSL interface is inherited from the MotionApps Platform, which is intended for use with an operating system. Therefore, each MLSLSerial function takes a `void *sl_handle` to pass a file handle.

On a typical embedded system, the `sl_handle` abstraction may often be omitted. In the provided AT32 MLSL, `sl_handle` is not used and is always passed `NULL`.

`inv_serial_open` should make the I^2C serial interface ready for use. The arguments `char *port` and `void **sl_handle` are always passed by user code (as opposed to code inside the Embedded MotionApps Platform) so they may be used as seen fit. If `sl_handle` is used, Inv_serial_open should allocate and initialize any required resources.

`inv_serial_reset` should reset the I^2C serial interface to an idle state and clear any error conditions in the I2C hardware peripheral or driver. It is only called by user code (not from inside the Embedded MotionApps Platform).

`inv_serial_close` should free any resources associated with the `sl_handle`, if applicable. It may make the I^2C serial interface unavailable for use. It is only called by user code (not from inside the Embedded MotionApps Platform).

4.4 Serial Read and Write

- `inv_error_t inv_serial_single_write(void *sl_handle, unsigned char slaveAddr, unsigned char registerAddr, unsigned char data)`
- `inv_error_t inv_serial_write(void *sl_handle, unsigned char slaveAddr, unsigned short length, unsigned char *data)`
- `inv_error_t inv_serial_read(void *sl_handle, unsigned char slaveAddr, unsigned char registerAddr, unsigned short length, unsigned char *data)`

`inv_serial_single_write` writes a single byte of data to an I2C device register. For more information on I2C device registers, see the IMU-3000 or MPU-6050 functional specification.
`inv_serial_write` is a generic multi-byte write to an I2C device. It does not assume the device follows a register convention. For multi-byte writes to a single I2C register (or set of sequential registers), the first byte in **`*data`** should be the register address.
`inv_serial_read` is a multi-byte read from an I2C device. It assumes a read from an I2C device register.

`inv_serial_single_write`

`void *sl_handle`

An operating system dependent handle and may be passed NULL if unneeded.

`unsigned char slaveAddr`

The I^2C slave address of the device. The LSB of the 7-bit I^2C address is aligned with the LSB of the unsigned char.

`unsigned char registerAddr`

The register address for the byte to be written.

`unsigned char *data`

A pointer to the data to be written.

`inv_serial_write`

`void *sl_handle`

As discussed above, this is an operating system dependent handle and may be passed NULL if unneeded.

`unsigned char slaveAddr`

The I^2C slave address of the device. The LSB of the 7-bit I^2C address is aligned with the LSB of the unsigned char.

`unsigned short length`

The number of bytes to write. Bytes will be written to successive registers on the IMU/MPU. Note that length is specified as a 16 bit value, but that register addresses only occupy an 8 bit address space. The Embedded MotionApps Platform will never pass a `length` of greater than 255.

`unsigned char *data`

A pointer to the data to be written.

inv_serial_read

`void *sl_handle`

An operating system dependent handle and may be passed NULL if unneeded.

`unsigned char  slaveAddr`

The I^2C slave address of the device. The LSB of the 7-bit I^2C address is aligned with the LSB of the unsigned char.

`unsigned char  registerAddr`

The register address from which data must be read.

`unsigned short length`

The number of bytes to be read. Bytes will be read from successive registers on the IMU/MPU. Note that length is specified as a 16 bit value, but that register addresses only occupy an 8 bit address space. The Embedded MotionApps Platform will never pass a `length` of greater than 255.

`unsigned char *data`

A pointer for the data that was read to be copied into.

4.5 Serial Read and Write DMP Memory

- `inv_error_t inv_serial_read_mem(void *sl_handle, unsigned char slaveAddr, unsigned short memAddr, unsigned short length, unsigned char *data)`
- `inv_error_t inv_serial_write_mem(void *sl_handle, unsigned char slaveAddr, unsigned short memAddr, unsigned short length, unsigned char *data)`

`inv_serial_read_mem` and **`inv_serial_write_mem`** take the arguments

`void *sl_handle`

As discussed above, this is an operating system dependent handle and may be passed NULL if unneeded.

`unsigned char  slaveAddr`

The I^2C address of the IMU/MPU device.

`unsigned short memAddr`

The DMP memory address for the first byte to be written to. Note that this value is an unsigned short because the DMP memory has a 16 bit address space.

`unsigned short length`

The length of the data to be sent, in bytes. Note that this value is an unsigned short because the maximum length of a single transfer is 256 bytes.

`unsigned char *data`

A pointer to the data to be written to or read from DMP memory.

Notes:

The DMP memory interface uses three registers on the IMU/MPU device. The 16-bit memory address is selected by two 8-bit registers (DMP Bank and DMP Start Address), and a third register (DMP Read/Write) provides sequential read/write access to the DMP memory using sequential I^2C read/write. The implementations provided for AT32 break the `memAddr` argument into Bank and Start Address, and perform single-byte writes to the Bank and Start Address registers.

The implementations of `inv_serial_read_mem` or `inv_serial_write_mem` provided for AT32 partition reads and writes into chunks no larger than SERIAL_MAX_TRANSFER_SIZE, defined by default as 128 in platform/include/mlsl.h. This makes it possible to implement reads and writes with an I^2C driver which only accepts an 8-bit value for transaction length (i.e. the driver allows a maximum length of 255 bytes, when a request to `inv_serial_read/write_mem` may require up to 256).

DMP memory is partitioned into 256 byte banks. Each bank may be read or written in sequential byte order, but sequential read/writes cannot cross bank boundaries. When writing a block of memory which crosses bank boundaries, separate calls to `inv_serial_read_mem` or `inv_serial_write_mem` must be used. The implementations provided for AT32 incorporate error checking for these conditions.

4.6 Serial Read FIFO

- `inv_error_t inv_serial_read_fifo(void *sl_handle, unsigned char slaveAddr, unsigned short length, unsigned char *data)`

`inv_serial_read_fifo` takes the arguments

`void *sl_handle`

As discussed above, this is an operating system dependent handle and may be passed NULL if unneeded.

`unsigned char  slaveAddr`

The I^2C address of the IMU/MPU device.

`unsigned short length`

The length of the data to be sent, in bytes. This value may be no more than FIFO_HW_SIZE, defined in platform/include/mpu3050.h (for MPU-3000, MPU-3050, and IMU-3000 devices) or platform/include/mpu6000.h (for MPU-6000 and MPU-6050 devices).

`unsigned char *data`

A pointer for data read from the FIFO.

Notes:

The implementation of `inv_serial_read_fifo` provided for AT32 partition reads and writes into chunks no larger than `SERIAL_MAX_TRANSFER_SIZE`, defined by default as 128 in platform/include/mlsl.h. This makes it possible to implement reads and writes with an I^2C driver which only accepts an 8-bit value for transaction length (i.e. the driver allows a maximum length of 255 bytes, when a request to `inv_serial_read_fifo` may require up to 512 or 1024).

4.7 MLOS: Operating System Layer

- `void inv_sleep(int mSecs)`
- `unsigned long inv_get_tick_count(void)`

The Motion Library Operating System layer (MLOS) library, inherited from the MotionApps Platform, provides the Embedded MotionApps Platform with timing information. A reference MLOS is provided in **platform/at32/mlos_at32.c**.

`inv_sleep` is a delay function, which takes an argument in units of milliseconds. For the UC3 Reference Implementation, it is implemented using the `delay_ms()` function provided by the AVR Software Framework (ASF).

`inv_get_tick_count` should return a counter which increments once every millisecond. For the UC3 Reference Implementation, it is implemented using the AVR32_COUNT system register, which ticks once per CPU clock cycle. The value of the count system register is scaled according to the system clock rate to convert the units to milliseconds.

A variety of other mutex and file handling functions are required by the MLOS header. These may be safely stubbed out on the UMPL platform. For reference stubs, see the file mlos_at32.c.

4.8 UStore IO: Nonvolatile memory input and output

- `inv_error_t inv_ustore_open(void)`
- `inv_error_t inv_ustore_close(void)`
- `inv_error_t inv_uload_open(void)`
- `inv_error_t inv_uload_close(void)`
- `inv_error_t inv_ustore_byte(unsigned char b)`
- `inv_error_t inv_ustore_mem(const void *b, int length)`
- `inv_error_t inv_uload_byte(unsigned char *b)`
- `inv_error_t inv_uload_mem(void *b, int length)`

The Ustore layer provides an input and output interface for calibration data. Calibration data is always assumed to be binary data, and is of fixed length based on the compile options of the Embedded MotionApps Platform. In the 2.0 release, the 9-axis build of the Embedded MotionApps Platform requires less than 100 bytes of nonvolatile storage for gyro, accel, temperature compensation, and magnetometer calibration values.

A reference Ustore IO layer written against the AVR Software Framework nvram library is provided in **ustore_nvram_uc3_io.c.**

A stub Ustore IO layer with comments describing implementation recommendations is provided in **ustore_stub_io.c.** Please see the Embedded MotionApps Platform Functional Specification for more information.

4.9 **Compilation Setup**

The Embedded MotionApps Platform requires different C code to be built to select support for either IMU-3000 or MPU-6050.

i. Accelerometer support: Embedded MotionApps Platform provides two accelerometer drivers in **umpl/lib/accel**. Compile **kxtf9.c** to support the Kionix KXTF9 auxiliary accelerometer with the IMU-3000. Compile **mpu6050.c** to support the MPU-6050 accelerometer.

ii. Magnetometer support: Embedded MotionApps Platform provides a driver for the AKM AK8975 magnetometer in **umpl/adv/akm**. Compile **ak8975.c** to support the AK8975 auxiliary accelerometer with the MPU6050, revision B or greater. This doesn't apply if a 6-axis Embedded MotionApps Platform package is used.

iii. DMP Support: The Embedded MotionApps Platform provides two DMP Default files in **umpl/lib**. Compile **dmpDefault.c** to support the IMU-3000. Compile **dmpDefaultMantis.c** to support the MPU-6050.

iv. Platform setup code: The Embedded MotionApps Platform provides two platform setup files in **mllite**.

- Use mldl_cfg_init_umpl_3050.c with the IMU-3000.
- Use mldl_cfg_init_umpl_6050a2.c with MPU6050 revision A, or any MPU6050 with 6-axis fusion.
- Use mldl_cfg_init_umpl_6050.c with the MPU6050 revision B. The MPU6050 revision B is required for 9-axis fusion.

A description of Embedded MotionApps Platform APIs is provided in the "Embedded MotionApps Platform Functional Specification".

NOTE: The ML Module in the Embedded MotionApps Platform Functional Specification includes description of APIs that return different types of sensor data. The inv_get_xxx_float() functions, for example, inv_get_quaternion_float, will soon be deprecated. Therefore the user is advised to refrain from calling these functions.

4.10 **Platform Specific Set-up**

As described in section 3.9, the platform specific setup is performed by a mldl_cfg_init_umpl file:

- mldl_cfg_init_umpl_3050.c
- mldl_cfg_init_umpl_6050a2.c
- mldl_cfg_init_umpl_6050.c

These files allocate and initialize the data structures used in the MotionApps Platform's driver layer. The most common items changed in each file are defined as macros at the top of the file for readability.

1. Gyroscope Orientation Matrix: The orientation of the sensor fusion solution is, by default, relative to the axes of the gyroscope. In that case, the gyroscope orientation matrix is the identity matrix. If the sensor fusion solution needs to be aligned to other axes, specify another orientation matrix by changing the GYRO_ORIENT_rc values. Matrix elements are specified by two digits indicating the row and column indexed starting at zero. Elements must have a value of 1, -1, or 0, and the matrix must specify a right-handed coordinate system. More information on the orientation matrix is provided in section 3.11.
2. Accelerometer Slave Description: The system accelerometer is specified by giving the get_slave_description function from the accelerometer driver file. The I^2C slave address of the device must also be given.
3. Accelerometer Orientation Matrix: The orientation of the accelerometer axes are specified relative to the gyroscope axes. On the MPU-6050, the accelerometer orientation matrix is the identity matrix. IMU-3000 based systems with an external accelerometer will have an orientation matrix dependent on the relative orientation of the auxiliary accelerometer. The values follow the same conventions as the gyroscope orientation matrix.
4. (Optional) Magnetometer Slave Description and Magnetometer Orientation Matrix: On systems compiled with UMPL_NINE_AXIS enabled, the magnetometer slave description and orientation matrix must be given as well. These values follow the same convention as the accelerometer slave description. The file mldl_cfg_init_umpl_6050.c allocates and sets up the additional structures required to specify the magnetometer data.

4.11 **Mounting Matrix**

The mounting matrix, also known as orientation matrix, contains information on how the sensors are mounted on PCB. This allows PCB designers flexibility to pick any orientation to mount the sensors.

When the mounting matrix is true to the sensor mount orientation, the rotation of sensor board along x, y and z axis will result in the sensor data having the right direction and polarity. In other words, the mounting matrix will swap the sensor axis and apply minus signs where ever necessary in order to rotate all the sensor data into a reference frame useful for the application. This way the same application code can run on systems with different PCB layouts by changing the mounting matrix alone. The orientation of x, y and z axis and its polarity is as shown in Figure 3.

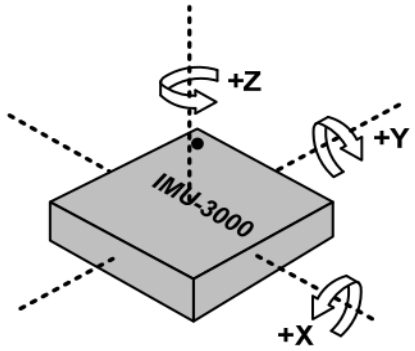

Figure 3: Sensor coordinate system

4.12 **Compile Flags**

Mandatory compile flags:

i. UMPL

This flag is required to build the Embedded MotionApps Platform.

ii. UMPL_ELIMINATE_ADVFUSION

This flag must be defined. It eliminates memory storage for structures not used in the Embedded MotionApps Platform.

iii. INV_FEATURE_GYROTC_UTC

This flag must be defined. It eliminates memory storage for structures not used in the Embedded MotionApps Platform.

iv. DMPDEFAULT_UNSORTEDKEYLOOKUP

This flag must be defined. It eliminates memory storage for structures not used in the Embedded MotionApps Platform.

v. INV_CACHE_DMP=0

This flag must be defined. It eliminates memory storage for structures not used in the Embedded MotionApps platform.

Sensor configuration specific compile flags:

i. CONFIG_MPU_SENSORS_MPU3050

This flag is required to build Embedded MotionApps Platform for MPU-3050.

ii. CONFIG_MPU_SENSORS_MPU6050B

This flag is required to build Embedded MotionApps Platform for MPU-6050 revision B or later.

iii. CONFIG_MPU_SENSORS_MPU6050A2

This flag is required to build the Embedded MotionApps Platform for MPU-6050 revision A.

iv. UMPL_NINE_AXIS

This flag is required to build the Embedded MotionApps Platform with 9-axis sensor fusion enabled. The MPU-6050 revision B or later and the AK8975 auxiliary magnetometer are required for 9-axis sensor fusion. The AKM AK8975 magnetometer calibration library "libak8975" is also required for magnetometer calibration. This library is not part of the Embedded MotionApps Platform and is available exclusively through AKM.

v. CONFIG_MPU_SENSORS_KXTF9

This flag indicates the Kionix KXTF9 auxiliary accelerometer driver will be used with the IMU-3000. The Kionix KXTF9 is the only auxiliary accelerometer supported with IMU-3000 at this time, so this flag is required to build the Embedded MotionApps Platform for IMU-3000. **This flag should not be used with the MPU-6050.**

vi. CONFIG_MPU_SENSORS_AK8975

This flag indicates the AKM AK8975 auxiliary magnetometer driver will be used with the MPU-6050. The AK8975 is the only auxiliary magnetometer supported with MPU-6050 at this time, so this flag is required to build the Embedded MotionApps Platform with UMPL_NINE_AXIS enabled.

Platform specific compile flags:

i. UMPL_TARGET_AT32

This flag is required to build Embedded MotionApps Platform for the Atmel AVR32 UC3 platform.

ii. BIG_ENDIAN

This flag needs to be included to build Embedded MotionApps Platform for a big-endian platform, such as the Atmel AVR32.

iii. UMPL_DISABLE_STORE_CAL

This flag may be included to disable calibration storage and must be included if the UStore IO layer is not implemented.

iv. UMPL_DISABLE_LOAD_CAL

This flag may be included to disable calibration loading and must be included if the UStore IO layer is not implemented.

5 Embedded MotionApps Platform AT32 Reference Implementation

5.1 Supported Hardware

i. Atmel UC3 Xplained

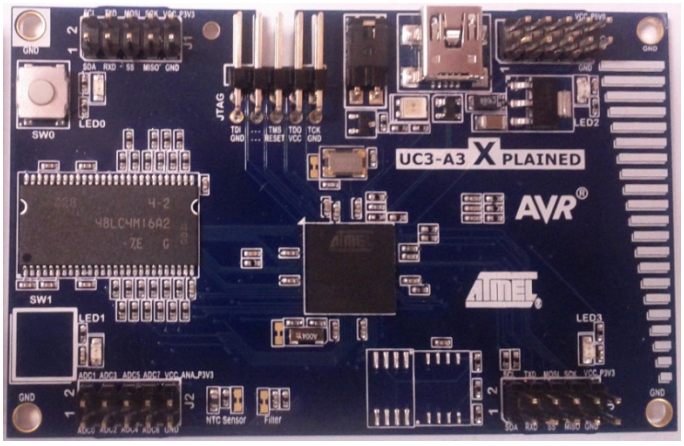

Figure 4: The Atmel UC3 Xplained Board

The Atmel UC3 Xplained development platform is available for sale at the Atmel website <link>.

An external programmer, such as the Atmel AVR JTAGICE mk II, is required to use the Atmel UC3 Xplained board with AVR Studio 5. More information on supported programmers is available on the Atmel website. The tutorial below uses the JTAGICE mk II.

ii. Inertial 2 Sensor Board

The Inertial 2 Sensor Board is show in Figure 5. The sensor board has a 3-axis InvenSense Gyroscope (IMU-3000), 3-axis Kionix Accelerometer (KXTF9-1026), 3 Axis Honeywell Magnetometer (HMC5883L). The Magnetometer is not being used by Embedded MotionApps Platform System.

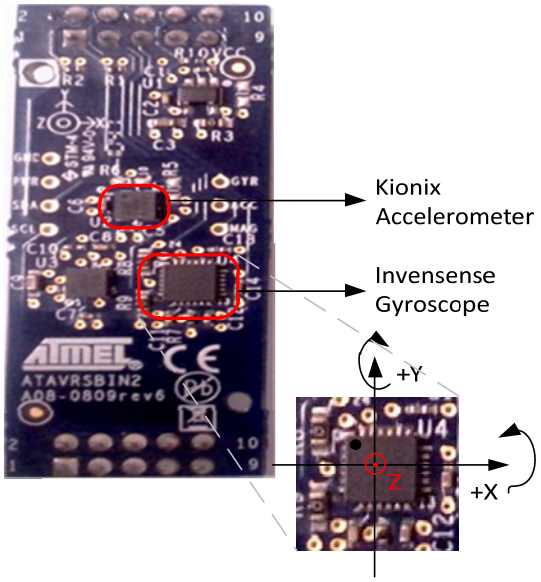

Figure 5: The Inertial 2 sensor board

5.2 **Supported Software**

The Embedded MotionApps Platform UC3 reference implementation has been built with AVR Studio 5 and the AVR Software Framework. Please visit the Atmel website to download the latest release of AVR Studio 5, which will includes an IDE, the AVR32 toolchain, and the AVR Software Framework.

At the time of this release, the AVR Studio 5 is still in beta. The Embedded MotionApps Platform UC3 reference implementation has been built with AVR Studio 5 Version 5.0.1038, and AVR Software Framework Version 2.3.0.

6 Tutorial: AT32 Reference Implementation with AVR Studio 5

6.1 Hardware Setup

Mount the Inertial 2 sensor board on the headers on the left side of the UC3 Xplained board, as shown in Figure 6.

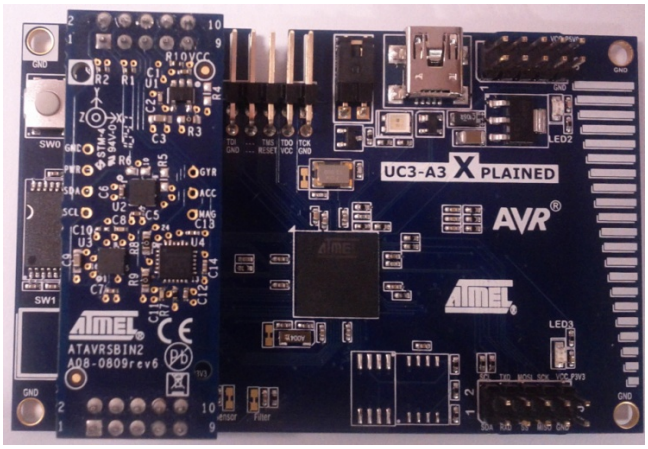

Figure 6: Xplained board with Inertial 2 board mounted

Connect the JTAG connector to your AVR programmer. Connect a USB cable to the mini-B USB port to provide power to the board.

6.2 Building Embedded MotionApps Platform Inertial 2 project in AVR Studio 5

Open the Embedded MotionApps Platform Inertial 2 project in AVR Studio 5.

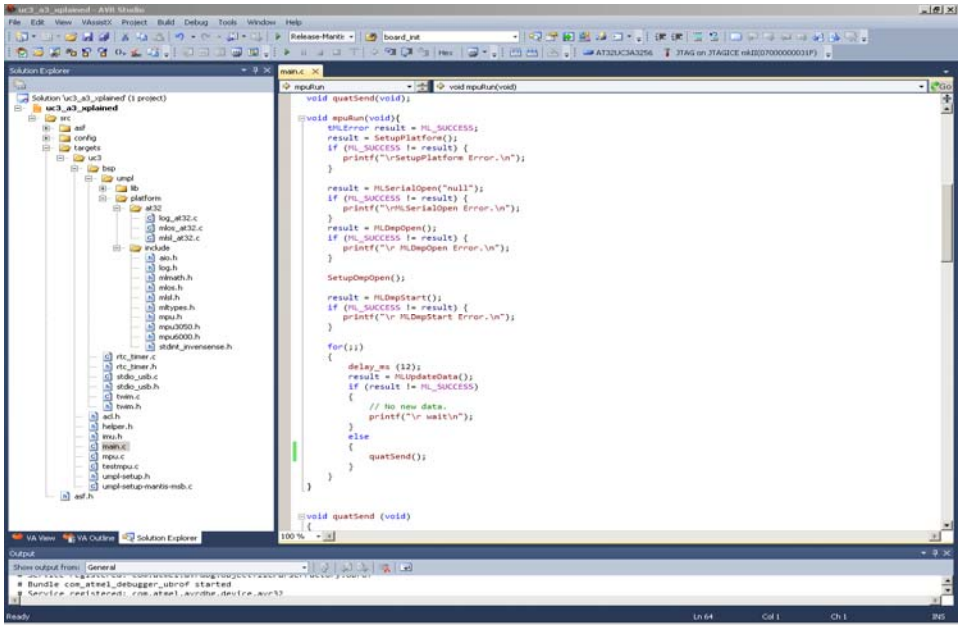

Figure 7: Solution Explorer view on AVR Studio 5

A project file tree will be available in the Solution Explorer as shown in Figure 7. The entry point for the firmware is in **targets/uc3/main.c**. Embedded MotionApps Platform files may be found under **targets/uc3/bsp/umpl/.**

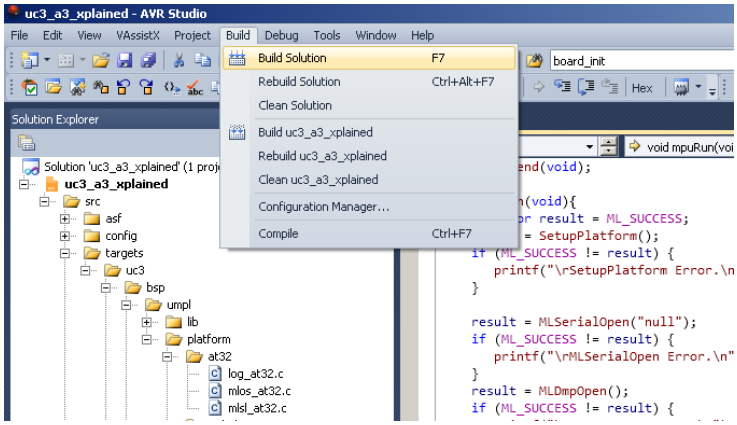

Figure 8: Building Solution

To build the Embedded MotionApps Platform demo solution, click on *Build -> Build Solution* as shown in Figure 8.

6.3 Programming the UC3 Xplained Board

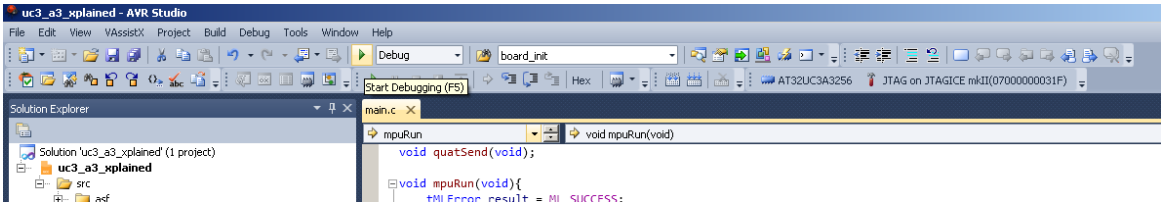

Figure 9: Debugging Solution

To program the UC3 Xplained Board, click on the green play button in the toolbar as shown in Figure 9.

6.4 COM Port Setup

After the UC3 Xplained Board has been programmed, the UC3 USB interface will be discovered by your PC as a CDC device.

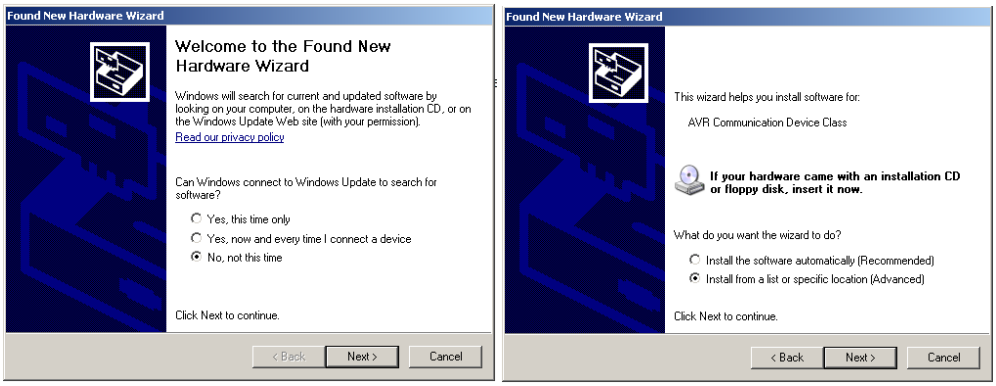

Figure 10: New Hardware Installation Wizard

Select “No, not this time”, then “Install from a list or specific location.”

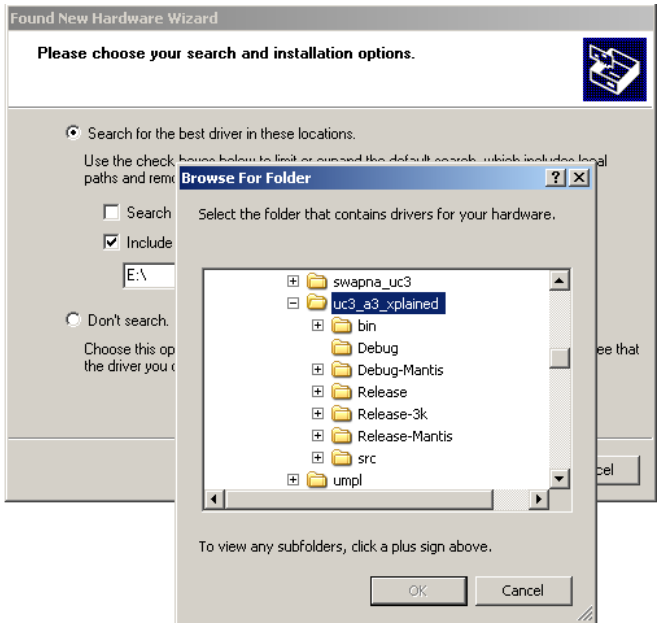

Figure 11: New Hardware Installation Wizard

Under “Include this location in your search”, browse to the Embedded MotionApps Platform Inertial 2 project directory.

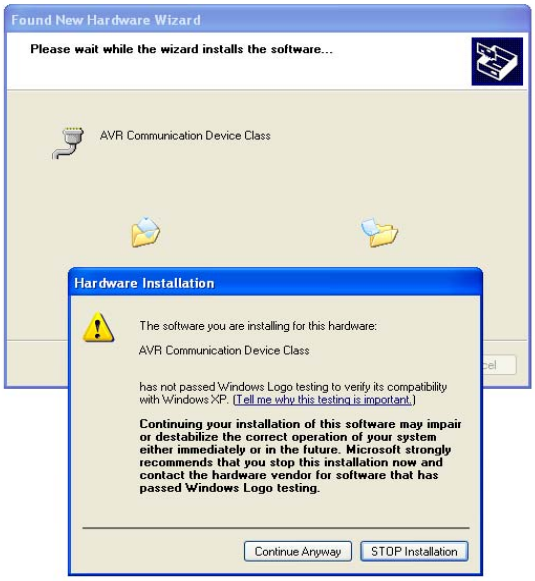

Figure 12: New Hardware Installation Wizard

The Found New Hardware Wizard will prompt you to approve the AVR Communication Device Class driver. Select “Continue Anyway”.

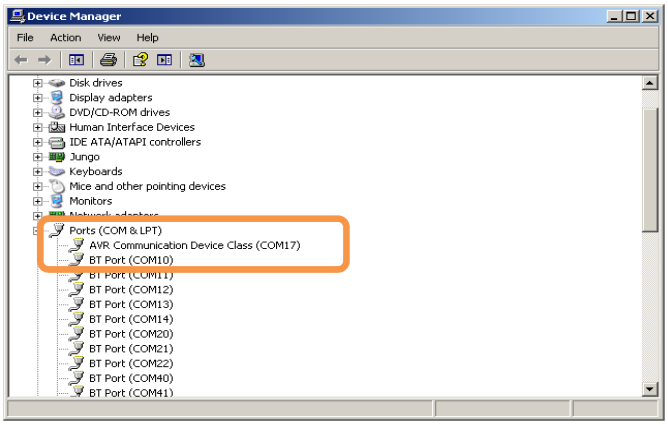

Figure 13: Device Manager showing AVR device

Once the driver has been successfully installed, the UC3 Xplained board will enumerate as an AVR Communication Device Class in the Device Manager.

To confirm the UC3 Xplained application is successfully communicating with your computer, you should open the UC3 COM port and inspect the serial stream.

1. Open HyperTerminal, or another terminal application. Select the UC3 COM port, as found in the Device Manager. For COM port settings, select 115200 bits per second, 8 data bits, No parity bits, 1 stop bit, and Hardware flow control.

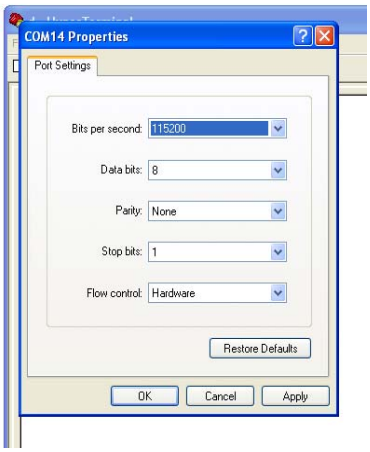

Figure 14: COM Port Settings

2. Once the connection to the COM port has been opened, press any key on the keyboard to initiate communication with the UC3 board. The UC3 application will begin sending binary data over the COM port. This data is separated into packets beginning with the dollar sign ($) and 14 bytes in length.

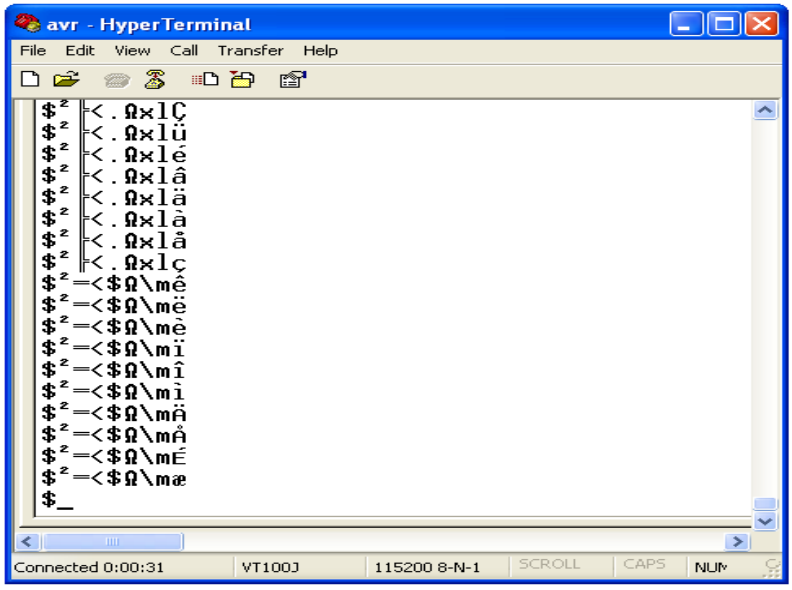

Figure 15: Binary data captured on serial terminal with application running

6.5 Running Demo Applications on UC3 Xplained Board

a. Tea Pot Application

To demonstrate Embedded MotionApps Platform, a “Tea Pot” application has been provided. This application was developed using Microsoft Visual Studio and the entire solution is included in the release package. Find the Tea Pot Application executable in the release folder. This application needs the COM port number of the AVR Xplained board as a DOS argument. Create a shortcut to the executable, right click on the shortcut, select properties, and add the COM port number in the target field. This is shown in Figure 16.

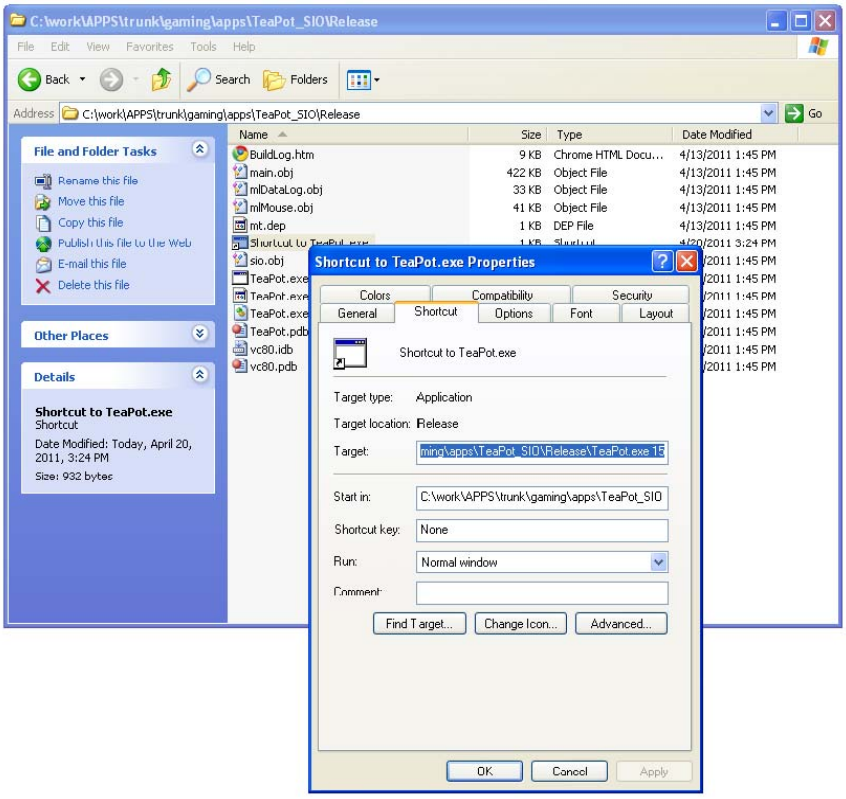

Figure 16: Adding COM port number to the shortcut of the application

Run the application by double clicking on the shortcut. The mouse pointer will now be controlled by the movement of the UC3 Xplained board. Also, a 3D teapot will be displayed on another window, as shown in Figure 17. Notice the orientation of the tea pot change according to the Embedded MotionApps Platform sensor fusion output. When the application is launched you will notice the pointer and teapot drift continuously. This is because the sensors are uncalibrated. Place the platform on a stable surface and do not move it till you see the drift stop.

A console window will be launched upon starting the tea pot application. Close the console window to exit the application.

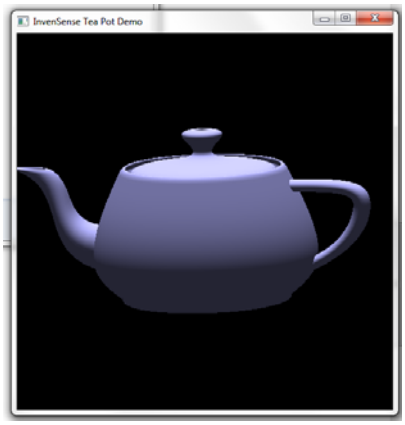

Figure 17. Tea Pot Application.

b. UC3 Debug Console

A debugging console application is provided to aid developers using the UC3 Xplained Embedded MotionApps Platform Reference Application. This application is provided as Python source code. It has the following functionality:

1. Provides a console for MPL_LOG messages.
2. Provides a simple cube demo for visualizing sensor fusion output. This cube demo provides identical functionality to the Tea Pot application from within the debug console.
3. Provides storage for calibration files. The UC3 reference application communicates with the debug console to load and save calibration data. This file will be saved and loaded from the file "calibdata" in the same directory as the debug console app.

Because the debug console is provided as Python source code, the user will need to install the following components to have a working Python environment.

1. Python 2.5.4 : The debug console was tested with Python 2.5.4 on Windows XP and Windows 7. Download the Windows Installer (python-2.5.4.msi) from http://www.python.org/download/releases/2.5.4/
2. Pyserial 2.5: The debug console requires pyserial to communicate on the serial port. Download the Python 2.x Windows Installer (pyserial-2.5.win32.exe) from http://pypi.python.org/pypi/pyserial
3. Pygame 1.9.1: The debug console uses the pygame library to render the cube demo. Download the Windows / Python 2.5 release (pygame-1.9.1release.win32-py2.5.exe) from http://www.pygame.org/download.shtml

You must install Python before installing the pyserial and pygame libraries.

You will want to add Python to your PATH for convenience. In Windows XP, right click on My Computer, select Properties, then in the Advanced tab, select Environment Variables. In the user variables (top section), select Path, click Edit, and add the directory C:\Python25 to the end of the list.

Once you have a working Python environment, you can run the debug console application from the command line with the command

```
python UC3umplDebugConsole.py <comport>
```

where `<comport>` is the UC3 COM port number.

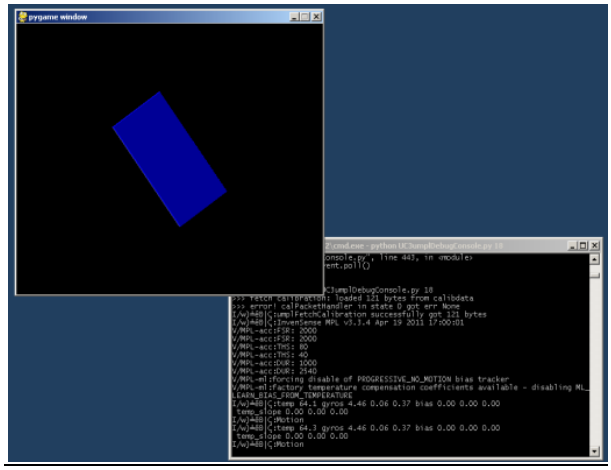

Figure 18: Cube Demo Application