NXP Sensor Fusion

NXP Sensor Fusion for Kinetis & LPC MCUs

Rev. 2.2 — 30 May 2017 **User guide**

Document information

Info	Content
Keywords	Sensor fusion, accelerometer, gyroscope, magnetometer, altimeter, pressure
Abstract	Provides full details on the structure and use of the NXP Sensor Fusion Library for Kinetis and LPC MCUs.

Revision history

Rev	Date	Description
2.0	20160812	This version of the user manual documents Version 7.00 of the NXP Sensor Fusion Library. The prior published version was Version 5.00 of the NXP Sensor Fusion Library. The kit has undergone major restructuring, and most of this document is new. This document was written by Michael Stanley and Mark Pedley.
2.1	20161103	Version 7.10 of the NXP Sensor Fusion Library has been updated to utilize new CMSIS drivers for I^{2}C and SPI now offered via the MCUXpresso Config Tool. The major change here is the addition of one extra parameter to the I^{2}C and SPI read and write functions. This has bubbled up to include the same extra parameter in the installSensor function, as well as modified setup for I^{2}C and SPI. These now support putting the MCU into low power mode while waiting for serial transactions to complete. FreeRTOS references have been updated to FreeRTOS 9.0.0. The magnetic calibration functions have been updated for improved convergence when strong soft iron distortion is present.
2.2	20170530	Version 7.20 of the NXP Sensor Fusion Library adds support for FRDM-KL25Z and LPCXpresso54114. User Guide text has been updated for consistency with the latest development environments.

Contact information

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

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

Sensor fusion is a process by which data from several different sensors are *fused* to compute something more than could be determined by any one sensor alone. An example is computing the orientation of a device in three-dimensional space. That orientation is then used to alter the perspective presented by a 3D GUI or game.

The NXP Sensor Fusion Library for Kinetis & LPC MCUs (also referred to as *Fusion Library* or *development kit*) provides advanced functions for computation of device orientation, linear acceleration, gyro offset and magnetic interference based on the outputs of NXP inertial and magnetic sensors.

Version 7.20 of the development kit has the following features:

- Full source code for the sensor fusion libraries
- IDE-independent software based upon the MCUXpresso Software Development Kit (SDK).
- The Fusion Library no longer requires Processor Expert for component configuration.
- Supports both bare-metal and RTOS-based project development. Library code is now RTOS agnostic.
- Optional standby mode powers down power-hungry sensors when no motion is detected.
- 9-axis Kalman filters require significantly less MIPS to execute
- All options require significantly less memory than those in the Version 5.xx library.
- Full documentation including user manual and fusion data sheet

The fusion library is supplied under a liberal BSD open source license, which allows the user to employ this software with NXP MCUs and sensors, *or those of our competitors*. Support for issues relating to the default distribution running on NXP reference hardware is available via standard NXP support channels. Support for nonstandard platforms and applications is available at https://community.nxp.com/community/sensors/sensorfusion.

This document is part of the documentation for NXP Sensor Fusion Library for Kinetis & LPC MCUs software. Its use and distribution are controlled by the license agreement in the Software licensing section.

1.1 Software licensing

1.2 Software features

Version 7.xx of the Fusion Library is delivered as part of the Intelligent Sensing SDK (ISSDK), which itself is included within various Kinetis and LPC SDKs generated via the MCUXpresso Config Tools (https://mcuxpresso.nxp.com). By *various*, we mean that ISSDK and the fusion library are released as part of the following board configurations: FRDM-KL25Z, FRDM-K22F, FRDM-K64F and LPCXPRESSO54114.

V7.xx was redesigned from the ground up for easy portability. It provides access to source code for all functions. Software features include:

- Fusion and magnetic calibration algorithms
- Programmable multi-rate sensor sampling
- Programmable sensor fusion rate
- Supported frames of reference include AEROSPACE, Android and Windows 8.
- Includes drivers for NXP motion sensors
- Minimum functions to learn.
  - Write sensor readings into one set of global structures using the addToFifo function.
  - Read fusion results from a different set of global structures
- Ability to compile and link any combination of standard algorithms
  - Accelerometer only (tilt)
  - Magnetometer only eCompass (vehicle)
  - Gyro only orientation (relative rotation)
  - Accelerometer plus magnetometer 6-axis eCompass
  - Accelerometer plus gyroscope orientation (gaming)
  - Accelerometer plus magnetometer and gyroscope (full 9-axis)
- NXP’s award-winning magnetic calibration software, which provides geomagnetic field strength, hard and soft iron corrections and quality-of-fit indication
- Optional standby mode powers down power hungry sensors when no motion is detected

- Directly compatible with the NXP Sensor Fusion Toolbox for Android and Windows (Fusion Toolbox). Example programs include predefined Bluetooth/UART interfaces compatible with the Fusion Toolbox.

1.3 Supporting documentation

1.3.1 Included in the kit

Included in the docs directory, mentioned in Section 1.2:

- This user guide
- NXP Sensor Fusion Library for Kinetis Data Sheet
- NXP Application Note AN5016, Rev. 2.0: Trigonometry Approximations
- NXP Application Note AN5017, Rev 2.0: Aerospace, Android and Windows 8 Coordinate Systems
- NXP Application Note AN5018, Rev. 2.0: Basic Kalman Filter Theory
- NXP Application Note AN5019, Rev. 2.0: Magnetic Calibration Algorithms
- NXP Application Note AN5020, Rev. 2.0: Determining Matrix Eigenvalues and Eigenvectors by Jacobi Algorithm
- NXP Application Note AN5021, Rev. 2.0: Calculation of Orientation Matrices from Sensor Data
- NXP Application Note AN5022, Rev. 2.0: Quaternion Algebra and Rotations
- NXP Application Note AN5023, Rev. 2.0: Sensor Fusion Kalman Filters
- NXP Application Note AN5286, Rev. 2.0: Precision Accelerometer Calibrations

1.3.2 Found elsewhere

- www.nxp.com/sensorfusion
- MCU on Eclipse blog at https://mcuoneclipse.com/
- Kinetis Design Studio software at www.nxp.com/kds
- Euler Angles at en.wikipedia.org/wiki/Euler_Angles
- Introduction to Random Signals and Applied Kalman Filtering, 3rd edition, by Robert Grover Brown and Patrick Y.C. Hwang, John Wiley & Sons, 1997
- Quaternions and Rotation Sequences, Jack B. Kuipers, Princeton University Press, 1999
- NXP Freedom development platform home page at nxp.com/freedom
- OpenSDA User’s Guide, NXP Semiconductors N.V., Rev 0.93, 2012-09-18
- NXP OpenSDA support page at nxp.com/opensda
- PE micro OpenSDA support page at www.pemicro.com/opensda
- Segger OpenSDA support page at https://www.segger.com/opensda.html

1.4 Requirements

1.4.1 MCU

Minimum requirements for the standard “all algorithms / no RTOS” build are:

- 40MHz ARM Cortex M0+ or higher MCU
- 60K bytes Flash NVM
- 12.5K bytes RAM
- 1 I^{2}C Port
- 1 UART compatible Port

MCUs that include floating point units will consume significantly fewer CPU cycles, although an FPU-less M0+ works fine.

In practice, the sensor fusion code has proven highly portable, and can be ported to almost any 32-bit MCU meeting the requirements above.

NXP markets a line of Arduino^{TM} compatible Freedom Development Boards which make an ideal platform for sensor fusion development projects. Figure 1 below illustrates the FRDM-K64F, which easily meets all of the requirements above.

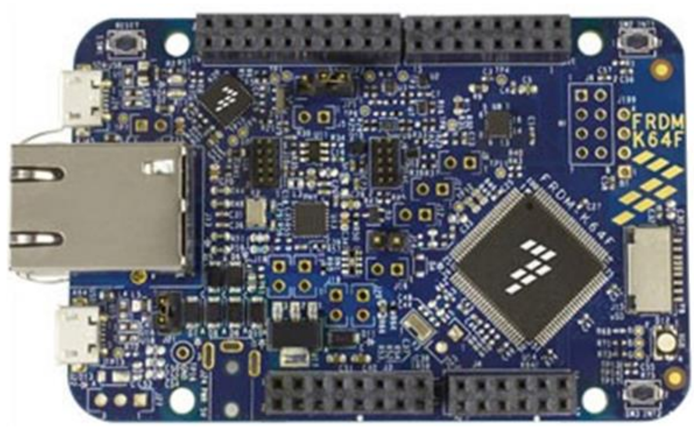


Fig 1. K64F Freedom Development Platform

NXP Freedom boards proven to be compatible on past releases of the toolkit include:

- **FRDM_KL25Z**
- FRDM_KL26Z
- FRDM_KL46Z
- FRDM_K20D50M
- **FRDM_K22F**
- **FRDM_K64F**
- FRDM_KV31F
- FRDM_KEAZ128
- **LPCXpresso54114**

The above is not an all-inclusive list; it simply represents the list of boards that the sensor fusion team at NXP has targeted in the past. The Version 7.20 sensor fusion offers out-of-the-box support for only the boards shown in bold.

Please consult http://www.nxp.com/freedom, the relevant MCU datasheet, and the NXP Sensor Fusion for Kinetis & LPC MCUs Datasheet for additional detail.

1.4.2 Sensors

The development kit supports sensors with both I^{2}C and SPI interfaces. Table 1 specifies the sensor types required on a per algorithm basis. The descriptions shown in the left column match those used in the Sensor Fusion Toolbox for Windows GUI. The second column includes the enabling bit-field as defined in build.h in your project.

Table 1. Sensor types required as function of algorithm

Algorithm	Related IfDef	Pressure	Accelerometer	Magnetometer	Gyro
Altitude	F_1DOF_P_BASIC	X			
Tiltmeter	F_3DOF_G_BASIC		X		
2D Automotive Compass	F_3DOF_B_BASIC			X	
Rotation	F_3DOF_G_BASIC				X
Tilt Compensated Compass	F_6DOF_GB_BASIC		X	X	
Gaming Handset	F_6DOF_GY_KALMAN		X		X
Gyro Stabilized Compass	F_9DOF_GBY_KALMAN		X	X	X

Many NXP Freedom development boards include one or more sensors. These are summarized in Table 2.

Table 2. Sensors by Freedom Development Platform

Board	Sensors
FRDM-KL02Z	MMA8451 accel
FRDM-KL05Z	MMA8451 accel
FRDM-KE02Z	MMA8451 accel
FRDM-KE06Z	MMA8451 accel
FRDM-KL25Z	MMA8451 accel
FRDM-K20D50M	MMA8451 accel
FRDM-KL26Z	FXOS8700 accel + mag
FRDM-K64F	FXOS8700 accel + mag
FRDM-K22F	FXOS8700 accel + mag
KV31F	FXOS8700 accel + mag
FRDM-KL46Z	MMA8451 accel + MAG3110
FRDM-KEAZ128	No Sensors

In addition, NXP has released a number of Freedom-compatible sensor boards specifically designed to support various algorithms shown in Table 1. These include:

- FRDM-FXS-9-AXIS (deprecated), which contains an FXOS8700CQ 6-axis accelerometer/magnetometer combination sensor and an FXAS21000 gyroscope
- FRDM-FXS-MULTI (deprecated), which contains all features of the FRDM-FXS-9-AXIS plus additional sensors
- FRDM-FXS-MULTI-B (deprecated), which contains all features of the FRDM-FXS-MULTI plus Bluetooth Module and battery

- FRDM-STBC-AGM01, which replaces the FRDM-FXS-9-AXIS. It contains FXOS8700CQ 6-axis accelerometer/magnetometer combination sensor and an FXAS21002 gyroscope
- The FRDM-FXS-MULT2-B replaces the FRDM-FXS-MULTI and FRDM-FXS-MULTI-B. It uses the newer FXAS21002 gyroscope, but it otherwise has the same sensor content.

The first three boards were built from the same PCB design and differ only in the number of parts on the platform. Those first three boards are no longer in production, but the Fusion Library works with any of the above boards. The FRDM-FXS-MULTI-B & FRDM-FXS-MULT2-B boards are the only ones that supports Bluetooth communications to the NXP Sensor Fusion Toolbox for Android.

Key components of the FRDM-FXS-MULTI-B and FRDM-FXS-MULT2-B are identified in Fig 2. The two boards have essentially the same layout. The major difference between the two is the new and improved FXAS21002 on the FRDM-FXS-MULT2-B.

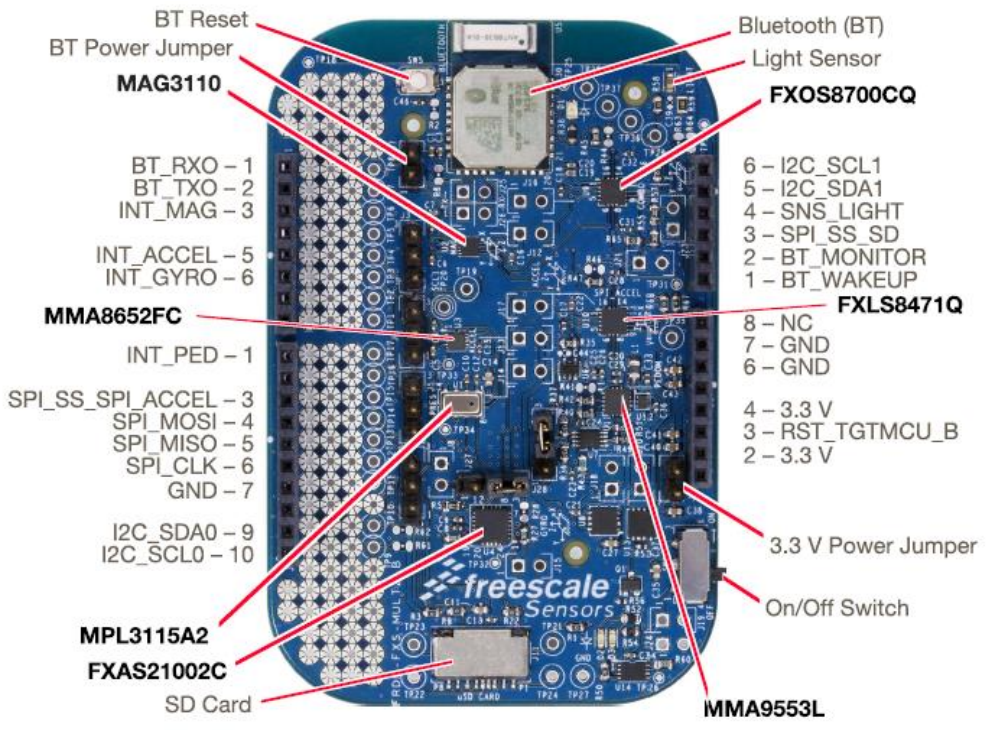


Fig 2. FRDM-Mult2-B with key components

Fig 3 shows the FRDM-FXS-MULT2-B Sensor Development Platform plugging into the FRDM-KL25Z Freedom Development Platform.

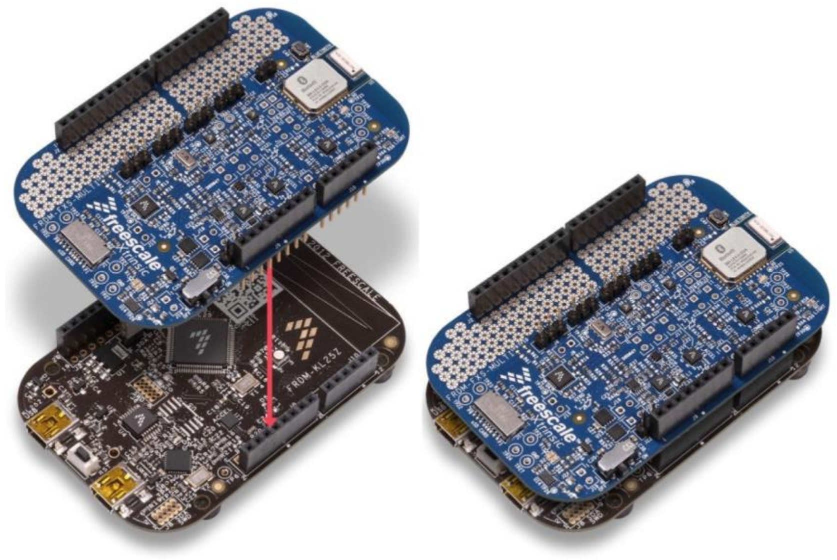


Fig 3. FRDM-FXS-MULTI-B mated to FRDM-KL25Z

Note: *When mating the two boards, check to ensure that the sensor board, when configured with a battery mounted on the back, is free of any obstacles presented by the base, NXP Freedom Development board. In particular, we have noted problems with FRDM-K22F boards. You will need to remove or relocate the battery.*

Note: The FRDM-FXS-MULT2-B (and its predecessor sensor boards) is NOT compatible with the LPCXpresso54114 MCU board*. Arduino pin assignments for the sensor and MCU boards are not compatible. The FRDM-STBC-AGM01 board IS compatible with the LPCXpresso51444.*

The FRDM-STBC-AGM01 is shown in Fig 4. It is a much simpler board than those shown above, but it still has all the components necessary to implement full 9-axis sensor fusion. It is compatible with all the base boards listed above**. Note J6 and J7 jumpers near the bottom of the board. These let you select between two different sources of I^{2}C signals. These must be set to the I2C1 settings for FRDM boards and the I2C0 settings for the LPCXpresso54114.**

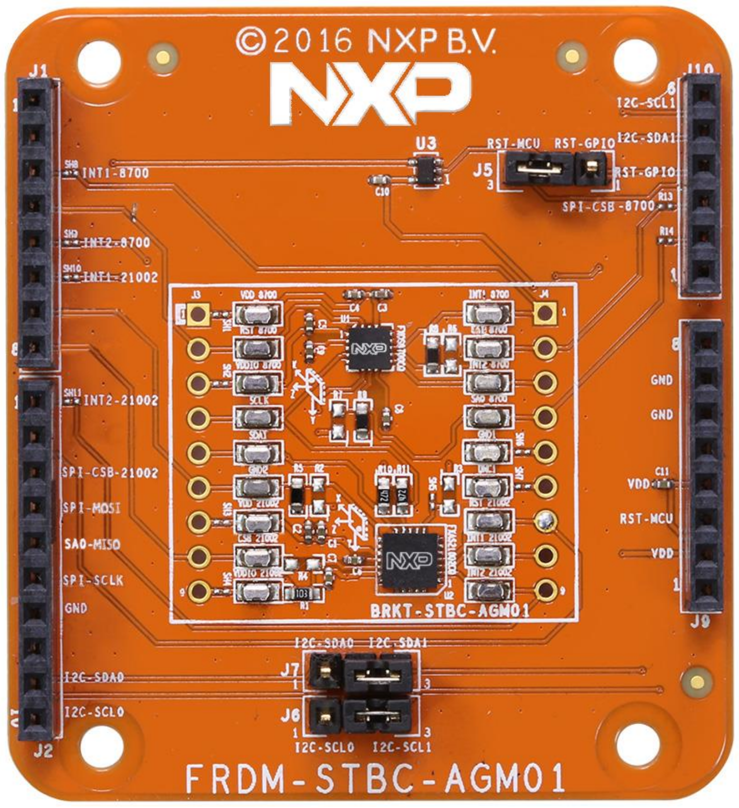


Fig 4. FRDM-STBC-AGM01 Freedom Development Platform

1.4.3 MCU Peripherals

Version 7.xx of the development kit represents a major departure from the design flow supported by Version 5.00 and earlier. Reference designs are no longer based on Processor Expert components, and instead rely on the MCUXpresso SDK. The software has been refactored to improve portability and support a more intuitive, object-oriented, interface. Accordingly, specific hardware requirements are associated with subsystems that will be explored in more detail in Section 3. A basic summary is:

Table 3. Primary subsystems and associated peripherals

Table description (optional)

Subsystem	Required	Peripherals Required	Comments
Main	Yes	Timer	PIT or similar if bare metal; SysTick commonly used for RTOS
Control	Debug minimum	UART	Example projects utilize two, one for OpenSDA wired connections, one for Bluetooth.
Status	No	Tri-Color LED	
ISSDK	Yes	I^{2}C and/or SPI	As required by the specific set of sensors used

Each project contains an include file called issdk_hal.h. This file pulls in header files for a Freedom board and sensor shield. When starting with a new board configuration, you should gather the schematics and user manuals for your board(s), then fill out a table similar to Table 4.

Table 4. FRDM-K64F and FRDM-FXS-MULT2-B / peripheral assignments

Subsystem	Function	Suggestion	Comments
Bare-metal main	Timer	PIT0	Encapsulated in driver_pit.c/.h
FreeRTOS main	Timer	SysTick	Possible interaction with Sensor Toolbox metrics measurement
Control	UART	UART3 on PTC17:16	Connects to Bluetooth module on sensor shields
		UART0 on PTB17:16	Connects to OpenSDA CDC[1] (UART/USB) on the Freedom board
Status	Red LED	PTB22	
	Green LED	PTE26	
ISSDK	I^2C	I^2C0 on PTC11:10	All I^2C sensors on FRDM-FXS-MULT2-B (excluding the FXLS8471Q)
	Accel/Mag	FXOS8700	I^2C address 0x1E on the FRDM-FXS-MULT2-B
	Gyro	FXAS21002	I^2C address 0x20 on the FRDM-FXS-MULT2-B
	Altimeter	MPL3115A2	I^2C address 0x60 on the FRDM-FXS-MULT2-B

1.4.4 IDE independence

As mentioned in the prior section, Version 7.xx of the development kit represents a major departure from the design flow supported by Version 5.00 and earlier. Reference designs are no longer based on Processor Expert components, and instead rely on the MCUXpresso SDK. CodeWarrior support has been dropped. IAR support has been added, and any screen shots shown in this user manual will generally be taken from IAR projects.

The fusion library itself is written entirely in C, and should be portable to any IDE.

The Kinetis Design Studio IDE is available for no cost at: http://www.nxp.com/kds.

The MCUXpresso IDE is available for no cost at http://www.nxp.com/mcuxpresso/ide.

1.4.5 Enablement tools

The demo sensor fusion builds include a serial command interpreter which can communicate sensor and fusion status back to Windows or Android-based graphical user interfaces. Using those same interfaces, the user can select which fusion algorithm to monitor and experiment with other fusion options. There are two versions of the GUI, which are described in the following two sections.

Both versions of the toolbox are supported by the same serial packet protocol which is encapsulated within the **control subsystem** shown in Fig 26. This subsystem can be omitted in your final product build, but we highly recommend that it be retained

1. CDC = Communications Device Class

throughout the product development/debug phases. *NXP support via the Sensor Fusion Community generally requires that the developer have access to the Sensor Fusion Toolbox for Windows.* Section 9 discusses many ways in which the visualization capabilities of this tool can help you to easily diagnose many implementation common problems.

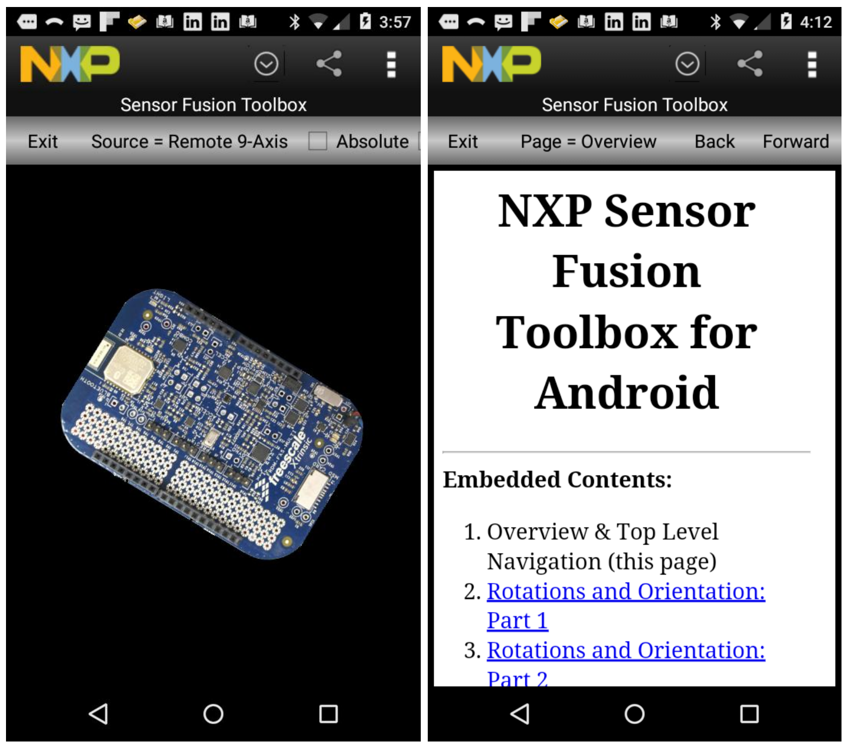
Both versions of the Sensor Fusion Toolbox are free.

1.4.5.1 Sensor Fusion Toolbox for Android

This is the original GUI first released to the world via Google Play in early 2013. The app is designed to communicate via standard Bluetooth with an NXP Freedom board equipped with a FRDM-FXS-MULT2-B sensor shield.

The application lets you visually explore sensor fusion tradeoffs. It also includes extensive sensor fusion tutorial information in the in-app documentation.

The latest version of this app may be downloaded at https://play.google.com/store/apps/details?id=com.sensors.fusion.



NXP Sensor Fusion Toolbox for Android

Embedded Contents:

1. Overview & Top Level Navigation (this page)
2. Rotations and Orientation: Part 1
3. Rotations and Orientation: Part 2

Fig 5. The Sensor Fusion Toolbox for Android

Select the Navigation icon and then **Documentation** to bring up the in-app help, shown on the right side of Fig 5.

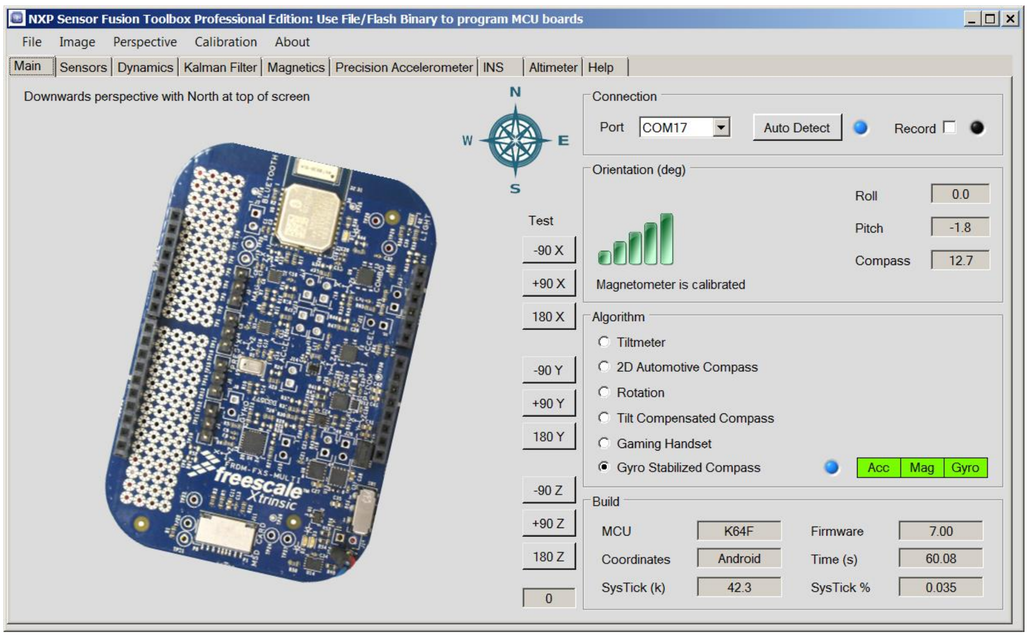
1.4.5.2 Sensor Fusion Toolbox for Windows

The Android version of the toolbox described above can provide a lot of insights into the behavior of the various sensor fusion algorithms in the toolkit, but it pales in comparison to the advanced features contained in the Windows version of the tool (Fig 6).

In addition to the Device view shown above, the Sensor Fusion Toolbox includes:

- Sensor outputs versus time graphs
- Fusion outputs versus time graphs
- Magnetic calibration sample constellation and calibration outputs
- Altimeter/temperature sensor tab
- Precision accelerometer calibration functions (new for this release)
- INS tab to play with double integration to approximate position (new for this release)
- The ability to save and restore accelerometer, magnetometer and gyroscope calibration coefficients for quick startup.

If your Windows PC has a Bluetooth interface or USB dongle, you can communicate wirelessly via Bluetooth to your FRDM-FXS-MULT2-B equipped Freedom board. If your shield does not include Bluetooth support, you can connect via USB to your PC. The interface controls are identical in both cases.



Downwards perspective with North at top of screen

Connection

Port COM17 Record

Orientation (deg)

Roll	0.0
Pitch	-1.8
Compass	12.7

Magnetometer is calibrated

Algorithm

- Tiltmeter
- 2D Automotive Compass
- Rotation
- Tilt Compensated Compass
- Gaming Handset
- [✓ Gyro Stabilized Compass]

Build

MCU	K64F	Firmware	7.00
Coordinates	Android	Time (s)	60.08
SysTick (k)	42.3	SysTick %	0.035

Fig 6. Sensor Fusion Toolbox for Windows

The Sensor Fusion Toolbox for Windows can be downloaded from http://www.nxp.com/sensorfusion.

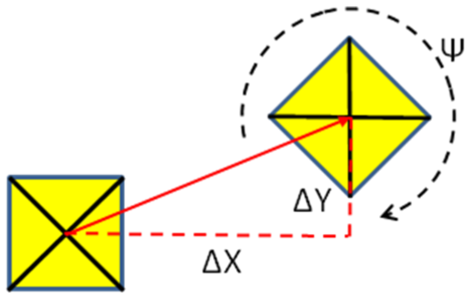
2. Sensor Fusion topics and options

If you are familiar with accelerometers, magnetometers, gyroscopes and frames of reference, you can skip Section 2.1. Likewise, if you are already an experienced user of prior versions of the Freescale/NXP sensor fusion library, you can skip all the way to Section 2.8, which addresses a topic new to Version 7.xx.

2.1 Vocabulary

If you consult any text on the dynamics of rigid bodies, you will quickly learn that any movement of any rigid body from point A to point B can be characterized as a translation plus a rotation.

Any movement from point A to point B can be decomposed into a translation plus optional rotation



In 3 dimensions, we need 6 DOF: ΔX, ΔY, ΔZ, ϕ, θ, ψ

Fig 7. Degrees of freedom explained

It takes six numbers to characterize that movement: change in X, Y and Z and rotations about X-, Y- and Z-axes. Notice that we are talking about the minimum set of numbers required to unambiguously specify a given movement. We are NOT talking about the number of sensors required to measure that movement.

So now, let’s talk about sensors. A basic 3-axis accelerometer returns values for linear acceleration in each of three orthogonal directions.

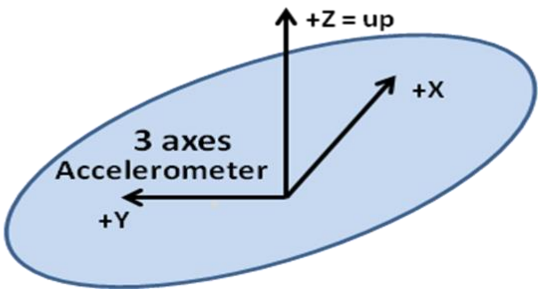


Fig 8. 3-Axis accelerometer

When you look at the figure, you can immediately see where the terms axis/axes come from. They refer to the sensor coordinate system axes.

There is an important thing you should consider about accelerometers at rest. When one of the axes associated with the sensor frame of reference is parallel to the gravity vector, as it is in the figure above, you will get no additional information from the other two acceleration numbers. They will both be zero, and you will be unable to tell if the accelerometer is rotated about the axis parallel to gravity.

The next device in our toolbox is the gyro, which returns rates of rotation about each of the three sensor axes. Notice that we are talking about sensor axes here. **As the sensor rotates, so does its frame of reference for the next measurement.**

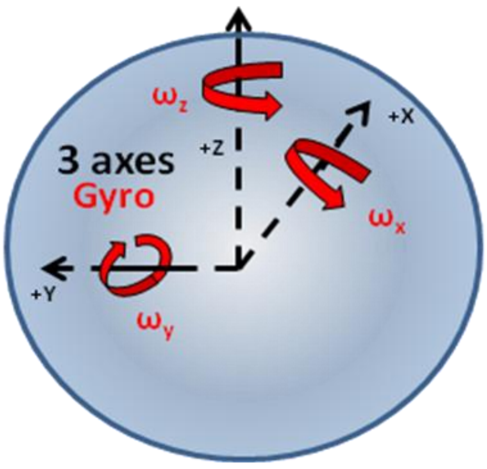


Fig 9. 3-Axis gyroscope returns the rotation rates about each of X-, Y- and Z-axes

A 3-axis magnetometer will return the X, Y and Z components of the ambient magnetic field. This is nominally the earth field for many applications, but readings may include significant offsets and distortions due to hard/soft iron effects. **The magnetometer is also subject to the same issue as an accelerometer – if one of the sensor axes is parallel to the ambient magnetic field vector, then the other two sensor axes will return values of zero.** The good news is that since the earth magnetic field and gravity are never collinear[2], between our accelerometer and magnetometer, we have enough information to figure out the current device orientation, regardless of how we rotate the sensor.

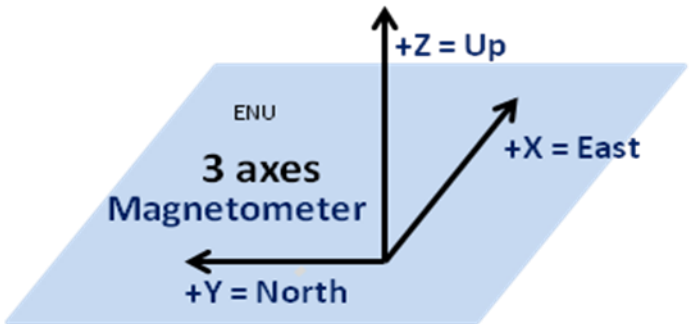


Fig 10. 3-Axis magnetometer will allow you to align yourself with the earth's magnetic field

Our first three sensors each returned a three-dimensional vector. But the pressure sensor returns just a single scalar value. Changes in pressure can be used to infer changes in altitude, which adds another source of information when computing vertical locations.

2. Except at the geomagnetic poles

Fig 11. Pressure[3] can be our 10th axis.

Combine an accelerometer with a gyro and you get a 6-axis Inertial Measurement Unit, or IMU.

Add a magnetometer to an IMU and you have a MARG (Magnetic, Angular Rate and Gravity) sensor. Add a compute engine to a MARG and you get an AHRS (Attitude and Heading Reference System).

A full 10-axis sensor subsystem = accelerometer + gyro + magnetometer + pressure

Use “DOF” when describing motion. Use “axis” or “axes” when describing sensor configurations.

2.2 3-Axis tilt

The 3-axis tilt algorithm is enabled with the **F_3DOF_G_BASIC** parameter in build.h. This algorithm requires only a single 3-axis accelerometer. It is only capable of modeling roll, pitch and tilt from vertical. It has no sense of compass orientation. This implies that the four orientations shown in Fig 12 will return exactly the same tilt value.

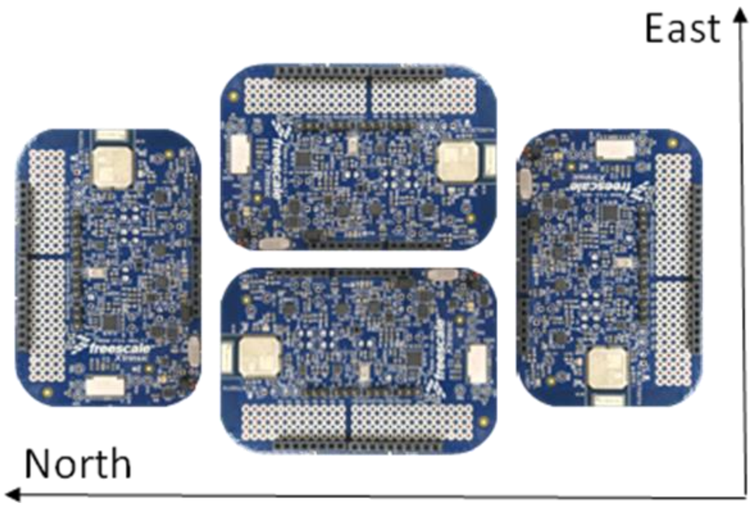


Fig 12. Equivalent tilt cases

This can result in some interesting behaviors when you invert the sensor board in the Sensor Fusion Toolbox. You may observe a “twist” by 180 degrees in the apparent orientation of the board. Both before and after are correct because of the ambiguity in the orientation representation.

3. The NXP MPL3115A2 can provide pressure or an estimate of altitude. It can also provide a temperature measurement.

2.3 2D automotive eCompass

This algorithm utilizes a 3-axis magnetometer to compute compass heading, while forcing the apparent orientation of the board to always be flat with respect to the earth.

For general use, the 6-axis tilt compensated eCompass discussed later is considered superior. But that algorithm is subject to errors due to linear acceleration. In contrast, the 2D automotive eCompass has no accelerometer, and therefore is immune to that issue. But it IS still sensitive to magnetic interference, and tilt will introduce errors in the apparent heading.

Tilt errors result from the fact that over most of the Earth’s surface, the magnetic field does not point to magnetic north, it points north and down. So tilting the sensor can rotate some of the up-down component of the field into the X-Y components. This then introduces errors in the heading computation.

This is easy to see using the Sensor Fusion Toolbox. Simply select the 2D algorithm option and tilt the board from horizontal.

2.4 3-Axis rotation

This option, which utilizes only a 3-axis gyroscope, is excellent for modeling rotations in three dimensions. But it has zero sense of up and down or compass heading. Orientation can be computed by integrating rotational rates over time, but if you do not know the initial orientation, you have no baseline for determining absolute orientation at any point in time.

2.5 6-Axis tilt-compensated eCompass

In Section 2.3, we noted that the 2D compass was subject to tilt error. When we utilize both an accelerometer and a magnetometer, we can mathematically align the sensor axes to the global earth frame of reference. This gives a full orientation estimate as well as accurate compass heading.

The limitations of this particular algorithm is that it is sensitive to magnetic interference as well as linear acceleration. Again, this is easy to see using the Sensor Fusion Toolbox. Wave a magnet a few inches from your board, or simply shake the board vigorously, and you will see the effects.

2.6 6-Axis gaming

This algorithm requires a 3-axis accelerometer and a 3-axis gyroscope. The gyro gives us the beautiful rotations we saw in the 3-axis rotation algorithm, and the accelerometer gives us an orientation with respect to gravity. What we do not get is compass heading. And this algorithm is still sensitive (although much less than the eCompass algorithm) to linear acceleration.

2.7 9-Axis

The 9-axis algorithm is generally considered to be the gold standard, as it does a good job of trading off the strengths and weaknesses of accelerometer, gyroscope and magnetometer to compute orientation with respect to the global earth frame. It also includes logic to identify magnetic and acceleration jamming, and ignore the affected sensor. The “Main” tab of the Sensor Fusion Toolbox for Windows includes a feature that identifies when magnetic jamming is detected.

You can still see some effects due to magnetic interference or linear acceleration, but they are largely attenuated. However, **they cannot be fully attenuated when those sources of interference are nearing steady state**. The algorithm takes advantage that we know how each of the 3 vectors in question (gravity, earth field and rotation) should change relative to the others when the board rotates. Introduction of constant offsets into assumed gravity and earth field vectors will throw off those calculations.

2.8 Inertial navigation – truth or fiction?

Accelerometers measure linear acceleration minus gravity. If you know the orientation of the accelerometer, you can subtract out the gravity component, leaving you with only linear acceleration. The linear acceleration vector can then be rotated from sensor to global frame of reference. Integrate that once and you get velocity, twice you get position. We all learned that in college. This is the basis for inertial navigation, AKA dead reckoning. These techniques have been in use since World War II.

BUT… those systems could afford to spend lots of money carefully calibrating sensors to remove offset and gain errors in the sensors. Today's MEMS and solid state sensors are mass manufactured with low cost in mind.

Let us consider a modern commercial MEMS accelerometer, with a specified post board mount accuracy of ±20 m*g* and an output noise level of 99 μ*g*/√Hz. Conservatively modeling that 20 m*g* solely in the X direction and with random noise distributed across all three axes, a simple Matlab simulation shows:

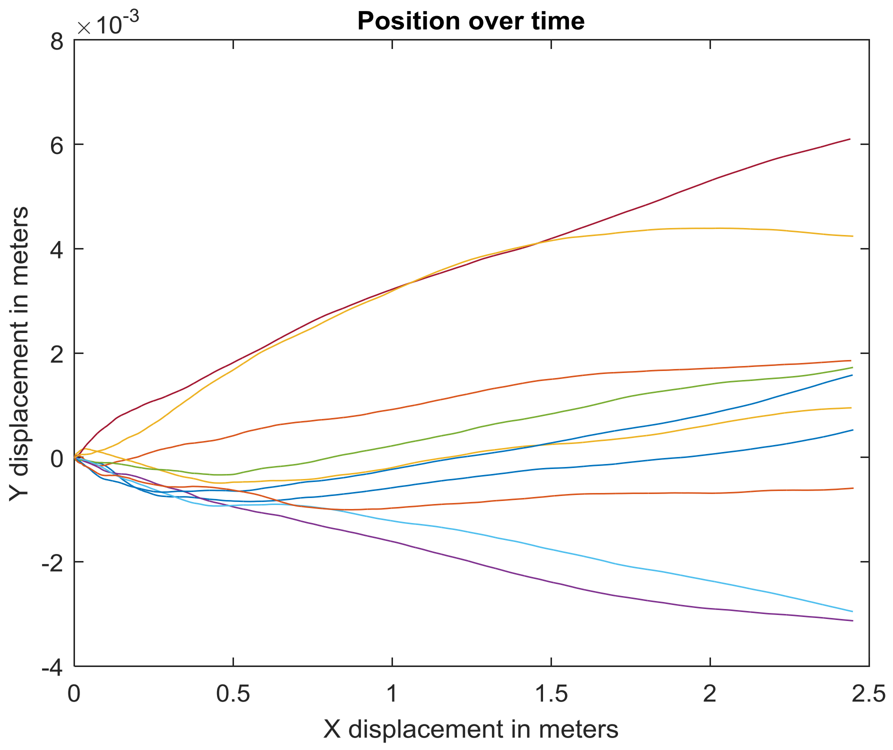


Fig 13. XY displacement over a 5 second interval (10 runs) for an uncalibrated accelerometer

The large displacement in X is due to the 20 m*g* offset in X. That 20 m*g* offset yields a velocity error which increases linearly with time, and a position error that grows quadratically.

The smaller displacement in Y is strictly due to wideband noise. Velocity errors in accelerometer noise will grow as a function of the square root of time. Position errors will grow as a function of time$^{2/3}$.

Doubly integrating even small input errors produces very large errors in time. Not understanding this basic math is one of the most common errors we see engineers new to the field making. If you would like to dig deeper, reference [9] provides a solid basis for understanding these relationships.

But what if we could calibrate out that 20 m*g* post-board mount offset? Version 7.xx of the sensor fusion library includes routines which allow you to do a one-time/one-temperature calibration which can reduce that 20 m*g* offset down to much lower levels. The same simulation used above, but with a value of 350 μg residual nonlinearity, yields the graph below.

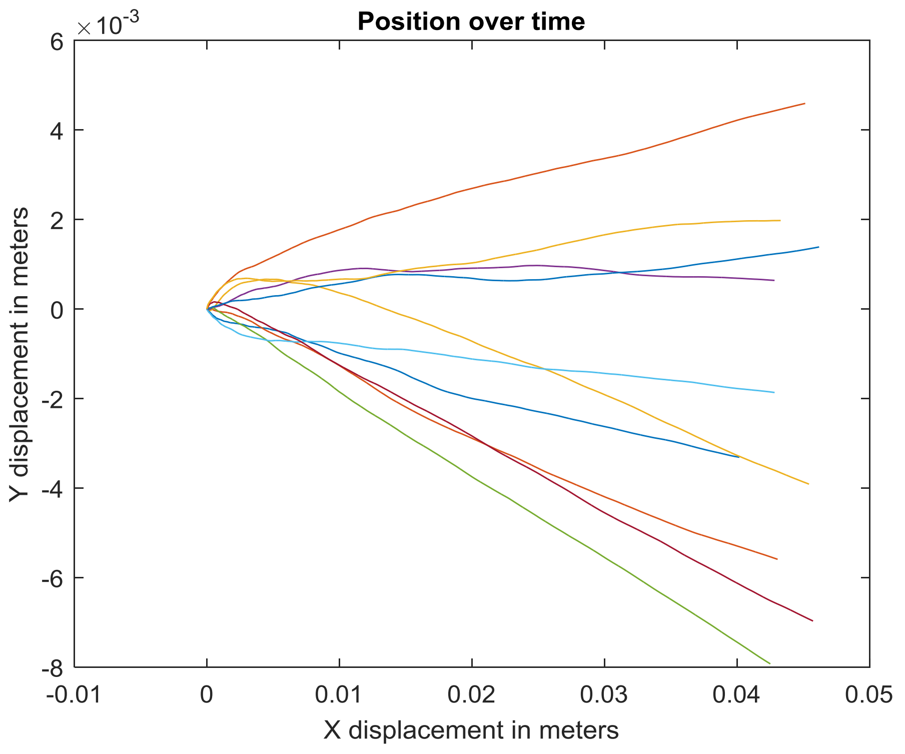


Fig 14. XY displacement over a 5 second interval (10 runs) for a calibrated accelerometer

Notice the dramatic improvement in the X-displacement, from a maximum of 2.5 meters to 5 cm. Bear in mind that these errors will continue to grow over time. But the point is that using inertial data to *aide* other systems (like GPS) is now possible so long as you do a reasonable job of precalibrating out known sensor errors.

The new “INS” tab in the Sensor Fusion Toolbox for Windows lets you experiment with errors in position over time. You will see very quick explosions in error for uncalibrated sensors. But as the discussion above shows, you will get better results after calibrating your sensors.

3. Quick Start Guides

3.1 Getting hardware

Hardware requirements were outlined in Sections 1.4.1, 1.4.2 and 1.4.3. No matter what MCU and sensor set you select for your final application, we recommend that you start with, and keep available, a known working solution based on the boards in those sections. When it comes time to debug your production design, you will find it useful to have that working example of sensor fusion as a reference.

The MCUXPRESSO SDK includes bare-metal and/or FreeRTOS sensor fusion projects for each supported MCU. Start with the sensor shield required by one of those projects.

1. Development boards can be purchased at http://www.nxp.com/freedom.
2. Visit http://www.nxp.com/opensda and install the appropriate OpenSDA drivers for your board. If desired, update the bootloader firmware on your board using the procedures outlined.

   Note: *Programming boards via the Sensor Fusion Toolbox require that the OpenSDA implementation on your embedded board supports drag and drop programming.*

3.2 Getting the MCUXPRESSO SDK

Version 2.xx of the MCUXpresso SDK is custom generated for each specific Kinetis and some LPC MCU. The MCUXpresso SDK contains only code and examples compatible with that MCU and the RTOS and tool options you select. You can configure and download your version at http://https://mcuxpresso.nxp.com. In the MCUXpresso SDK, you can expect to see a directory tree similar to:

- SDK_2.x_FRDM-K64F
  - boards
    - frdmk64f
      - demo_apps
      - driver_examples
      - …
    - frdmk64f_agm01
      - frdm_k64f.c
      - frdm_k64f.h
      - frdm_stbc_agm01_shield.h
      - issdk_hal.h
      - RTE_Device.h
      - issdk_examples
        - algorithms
          - sensorfusion
        - sensors
          - fxas21002
          - fxos8700
    - frdmk64f_mult2b
      - frdm_k64f.c
      - frdm_k64f.h
      - frdm_stbc_mult2b_shield.h
      - issdk_hal.h
      - RTE_Device.h

- issdk_examples
  - algorithms
    - sensorfusion
  - sensors
    - fxas21002
    - fxos8700
    - fxls8471q
    - mag3110
    - mma865x
    - mpl3115
- CMSIS
- devices
  - MK64F12
    - drivers
    - iar
- docs
- middleware
  - issdk_1.0
    - algorithms
      - sensorfusion
        - docs
        - sources
    - sensors
      - fxas21002.h
      - fxos8700.h
      - fxls8471q.h
      - mag3110.h
      - mma865x.h
      - mpl3115.h
- rtos
  - freertos_9.0.0 (version may vary)

SDK elements highlighted in red are utilized by the sensor fusion examples.

See Section 1.3.1 for supporting documentation included in the sensorfusion/**docs** directory. The **issdk_examples/algorithms/sensorfusion** directory includes example projects for FreeRTOS-based and/or bare metal applications. You can expect to see example projects for KDS, Atollic and IAR.

3.2.1 The Board/Shield Concept

ISSDK has been architected with the assumption that initial development will be done on NXP Arduino compatible MCU boards coupled with Arduino compatible sensor shields. Signals connecting the two boards are labeled A0 through A5 and D0 through D15. D14 and D15 correspond to the standard Arduino I2C pins. Each ISSDK project includes a <boardname>.h and <shieldname>_shield.h file which define hardware connections in terms of these standard signals. There is also a file called issdk_hal.h, which links the two for a specific project.

You will find these files under <sdk_root>/boards/<baseboard>_<shield>. If you develop your own hardware use the existing boards as templates to model communications within your system.

The FRDM-KL25Z SDK contains (for SF Version 7.20 forward) a “virtual shield” that can be used to compile your project. The virtual shield is a logical superset of a number of NXP sensor boards (notably FRDM-FXS-MULT2-B and FRDM-STBC-AGM01). Check the .h signal assignments against your hardware, and modify as necessary.

3.3 Compiling binaries

The development kit is compatible with any number of different development environments. Additional information on software and tool options for Kinetis & LPC MCUs can be found at Software and Tools for Kinetis MCUs.

The Kinetis Design Studio IDE can be downloaded for free from http://nxp.com/kds.

The MCUXpresso IDE can be downloaded for free from http://www.nxp.com/mcuxpresso/ide.

Sample projects are completely preconfigured. You should only have to:

1. Open the supply project(s) in the IDE of your choice
2. Compile
3. Download into your board

3.4 Microsoft Windows

You do not need to do any program development to start experimenting with sensor fusion algorithms on your board. The Sensor Fusion Toolbox for Windows comes with a variety of prebuilt binaries that can be installed on your development board.

The Sensor Fusion Toolbox for Windows can be downloaded from http://www.nxp.com/sensorfusion. The tool is free and is compatible with all the fusion features discussed in this user guide.

Assuming you already have a Freedom board and sensor shield in hand:

1. Connect Freedom board to PC using a USB cable
2. Start the Sensor Fusion Toolbox

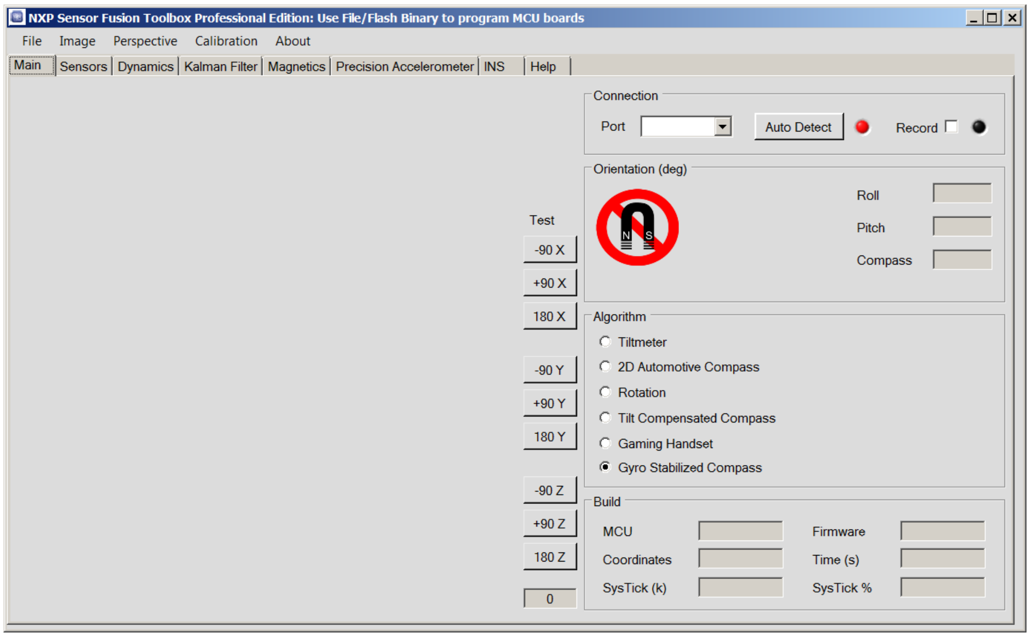
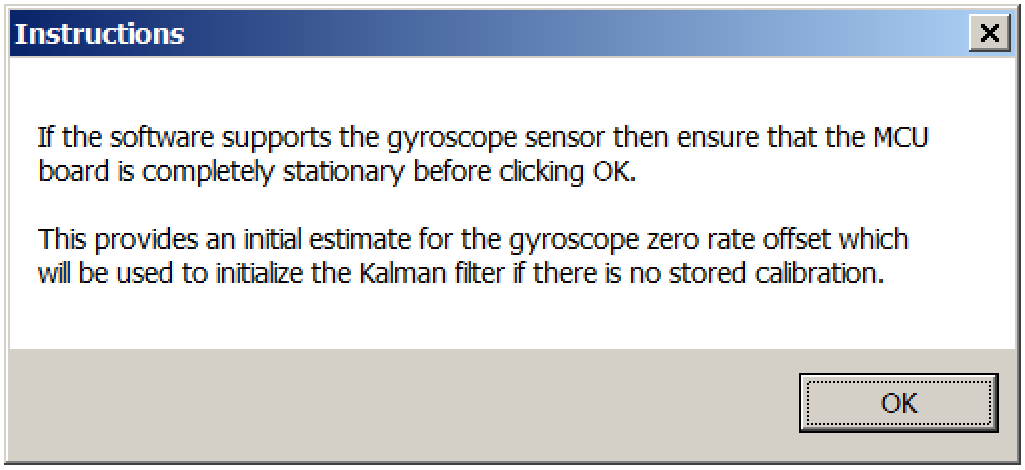


Fig 15. Sensor Fusion Toolbox For Windows at Startup

3. Program your board:
   a. Click File->Flash Kinetis Binary-><freedom board name>-><shield name>-><sensor_combo>.
   b. Check your USB connection as per the resulting dialog box and click OK after reading the full set of directions
   c. Select your board (MBED if using the ARM CMSIS-DAP bootloader) in the "Save As" dialog and click "Save"
   d. Click OK when informed that the board flashed.
   e. unplug and plug the board back in
4. Click the **Auto Detect** button. If your board is recognized, you will see the following dialog.


Instructions

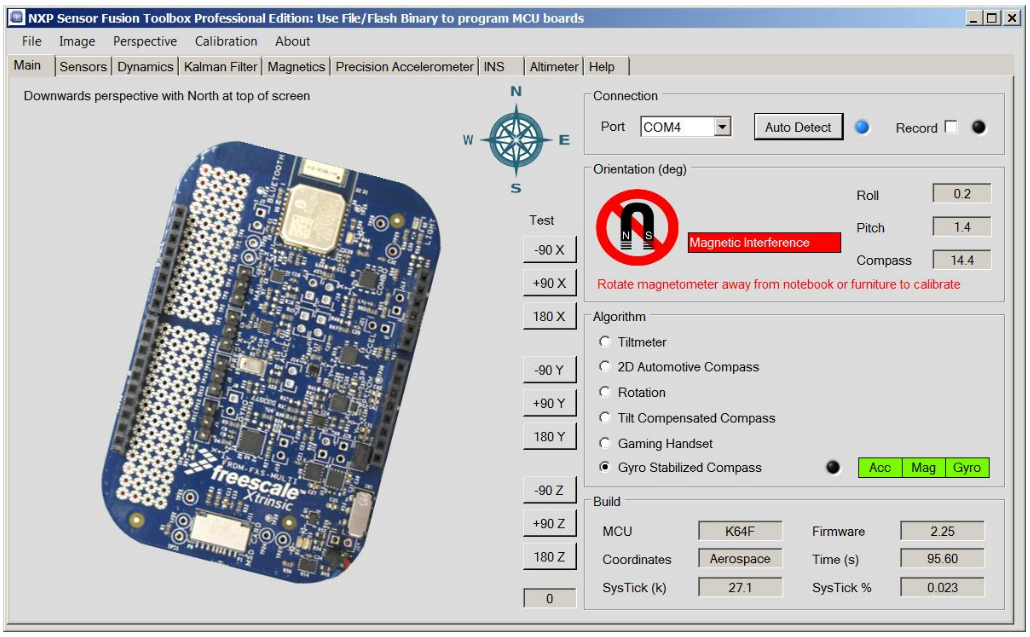
If the software supports the gyroscope sensor then ensure that the MCU board is completely stationary before clicking OK.

This provides an initial estimate for the gyroscope zero rate offset which will be used to initialize the Kalman filter if there is no stored calibration.

OK

Fig 16. Get ready to calibrate your gyroscope

Put your board flat on a tabletop before hitting OK. This will allow the board discovery process to also measure stationary gyro offsets. These are then used to initialize Kalman filter gyro offset values. Next you should see



Connection: Port COM4, Record ☐

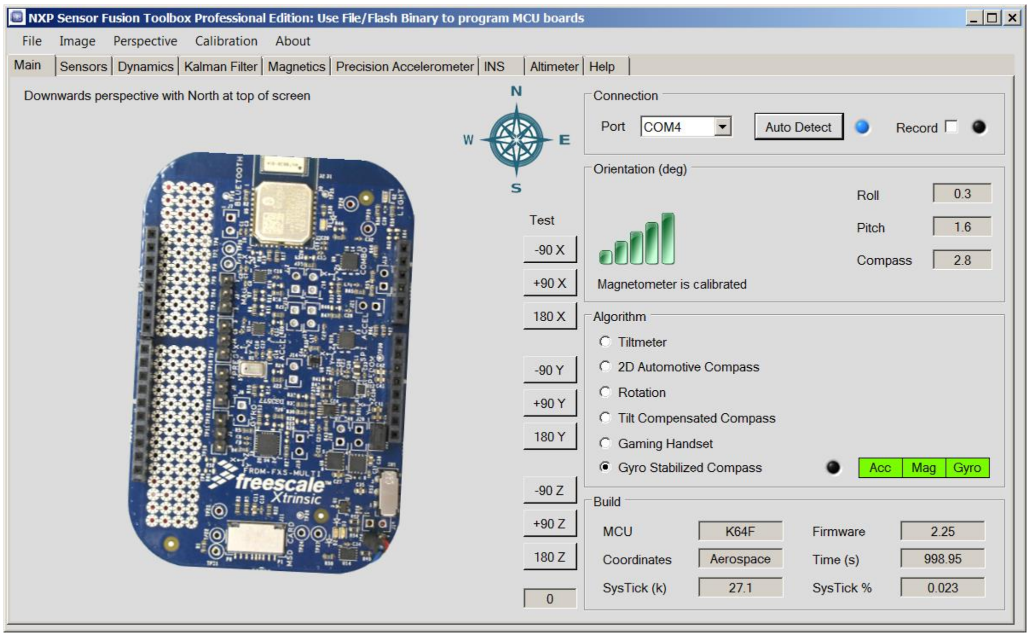
Orientation (deg): Roll 0.2, Pitch 1.4, Compass 14.4

Algorithm: ○ Tiltmeter ○ 2D Automotive Compass ○ Rotation ○ Tilt Compensated Compass ○ Gaming Handset [✓ Gyro Stabilized Compass]

Build: MCU K64F, Firmware 2.25, Coordinates Aerospace, Time (s) 95.60, SysTick (k) 27.1, SysTick % 0.023

Fig 17. Sensor Fusion Toolbox For Windows at startup

5. Pick up your board and rotate it in space, away from any objects that contain ferrous materials, such as furniture and computers, until the Magnetic Interference notice disappears.



Connection: Port COM4, Record ☐

Orientation (deg): Roll 0.3, Pitch 1.6, Compass 2.8

Algorithm: ○ Tiltmeter ○ 2D Automotive Compass ○ Rotation ○ Tilt Compensated Compass ○ Gaming Handset [✓ Gyro Stabilized Compass]

Build: MCU K64F, Firmware 2.25, Coordinates Aerospace, Time (s) 998.95, SysTick (k) 27.1, SysTick % 0.023

Fig 18. Sensor Fusion Toolbox For Windows after Magnetic Calibration

At this point, you have a working board that is properly communicating to the Sensor Fusion Toolbox. You can now start experimenting with the features listed in Section 1.4.5.2.

3.5 Android

The Sensor Fusion Toolbox for Android communicates via a development board via Bluetooth. Supported shield boards are the FRDM-FXS-MULTI-B and FRDM-FXS-MULT2-B boards. The former is no longer in production, having been replaced by the latter. The two boards look almost identical.

1. Program your development board using the standard sensor fusion demo (see Section 3.3).
2. The Sensor Fusion Toolbox for Android can be directly downloaded from Google's Play Store at https://play.google.com/store/apps/details?id=com.sensors.fusion.

   Alternately you can search for [NXP sensor fusion] in Google Play.

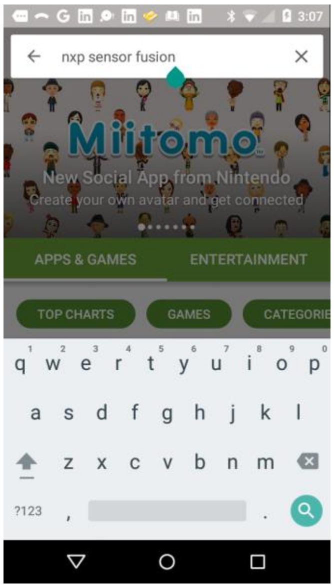


Fig 19. Searching for the App in Google's Play Store

If you search instead of using the direct link above, you may see two versions listed. The top version (with the diamond chip logo) is the legacy Freescale version of the tool. It will not be maintained going forward.

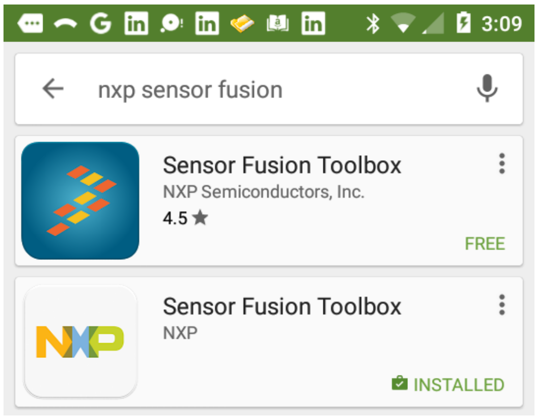


Fig 20. Sensor Fusion Toolbox For Android options

3. You should select the icon with the white and pastel NXP logo. It has all the features of the original, and will be the Android platform going forward.
4. Install the application in the normal manner.
5. Power up your sensor board. Ensure that the power jumper next to the Bluetooth module is installed (“A” in Fig 21).
6. FRDM-FXS-MULTI-B and FRDM-FXS-MULT2-B boards utilize Bluetooth modules from BlueRadios. Note the last six digits on the second line of the radio module (“B” in Fig 21).

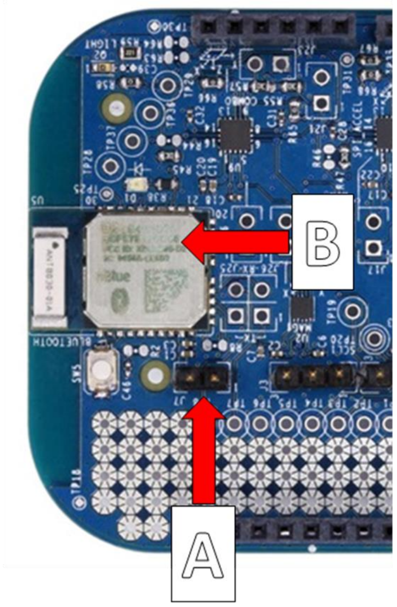


Fig 21. Key features on the FRDM-FXS-MULT2-B

7. On your Android device, select “Settings->Bluetooth” and search for a device beginning with “BlueRadios” and ending with the digits you noted above. Pair your Android device with that device.
8. Exit Settings
9. Start the application by clicking the “NXP” logo on your device.

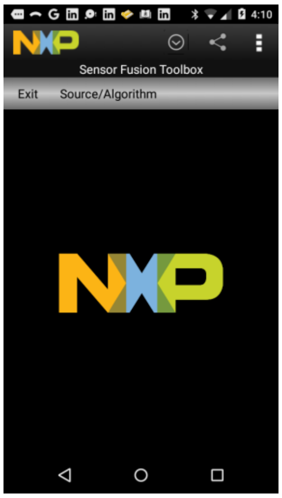


Fig 22. Startup screen when no paired device is found

10. Select the options menu in the upper right and choose “Preferences”

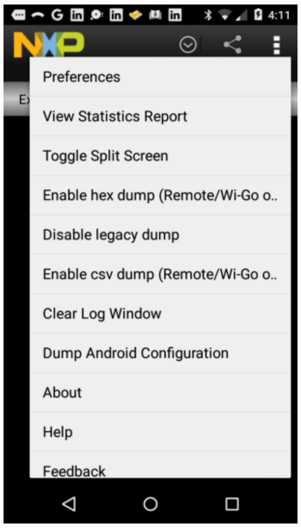
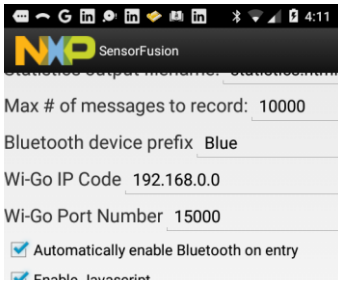


Fig 23. Sensor Fusion Toolbox for Android Options menu

11. Check the checkbox labeled “Automatically enable Bluetooth on entry”.



Max # of messages to record: 10000
Bluetooth device prefix Blue
Wi-Go IP Code 192.168.0.0
Wi-Go Port Number 15000
☑ Automatically enable Bluetooth on entry

Fig 24. Key features on the FRDM-FXS-MULT2-B

12. Click the **Save and Exit** button, then exit the program and restart it.

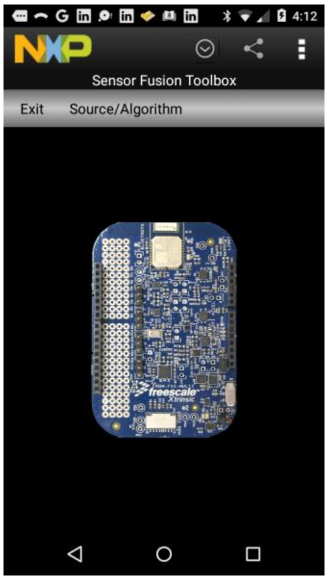


Fig 25. Key features on the FRDM-FXS-MULT2-B

13. Click **Source/Algorithm** and select **Remote 9-axis**. The PCB displayed on your Android device should track the physical board.

The Android version of the Sensor Fusion Toolbox includes extensive documentation and tutorial information. Select the Navigation icon and then Documentation to explore your options.

3.5.1 iOS

Unfortunately, NXP does not offer the Sensor Fusion Toolbox for iOS.

3.6 Converting a FreeRTOS Project to Bare Metal

It is really easy to convert FreeRTOS projects to bare metal. Here's the process used to convert the FreeRTOS AGM01 project to bare metal.:

- Download either FRDM-KL25Z, FRDM-K64F, FRDM-K22F or LPCxpresso54114 SDK with ISSDK and FreeRTOS checked in the MCUXpresso configuration screen.
- Navigate down the sensor fusion projects folder (<sdk_root>/boards/<board>_agm01/issdk_examples/algorithms/sensorfusion)
- Copy the freertos_agm01 to baremetal_agm01
- Open the copied project in your IDE of choice
- Remove FreeRTOSConfig.h from the sources folder
- Remove the entire freertos folder
- Replace main_agm01_freertos_two_tasks.c with main_baremetal.c (which can be found under <sdk_root>/middleware/issdk_1.0/algorithms/sensorfusion/sources)
- Remove the drivers/freertos folder
- Build and download

That's it!

4. Architecture

4.1 High level overview

Version 7.xx of the sensor fusion library represents a major departure from Version 5.00 and prior kits. The software has been repartitioned to localize hardware interfaces into a small number of subsystems with well-defined interfaces. This makes it possible for developers to retarget the library to entirely different MCUs, sensors and tool chains simply by swapping out the various subsystems and drivers.

Fig 26 illustrates the overall architecture. At the bottom of the stackup is the MCUXpresso Software Design Kit. The SDK can be freely downloaded from https://mcuxpresso.nxp.com/.

The MCUXpresso SDK is a collection of comprehensive software enablement for NXP Kinetis (and some LPC) Microcontrollers that includes system startup, peripheral drivers, USB and connectivity stacks, middleware and real-time operating system (RTOS) kernels. The SDK also includes getting started and API documentation along with usage examples and demo applications designed to simplify and accelerate application development on Kinetis & LPC MCUs.

All software is provided free-of-charge as assembly and C source code under permissive and open-source licensing. Support is provided through the MCUXpresso SDK Community Forum.

On top of the MCUXPRESSO SDK, we have:

- The status subsystem, which provides a simple visual indicator of the system status. The default implementation assumes an RGB LED driven via GPIOs.
- The control subsystem provides an interface to/from the NXP Sensor Fusion Toolbox. This is via UART serial port communication.
- The ISSDK provides a CMSIS compliant set of low level APIs for accessing sensor registers via I2C and/or SPI. All sensor drivers are written in terms of 6 ISSDK functions.
- Flash memory storage can be used to store/retrieve sensor calibration parameters on Kinetis devices. Other MCU families can swap this interface (which has only one function) out to support external EEPROM or other NVM options.

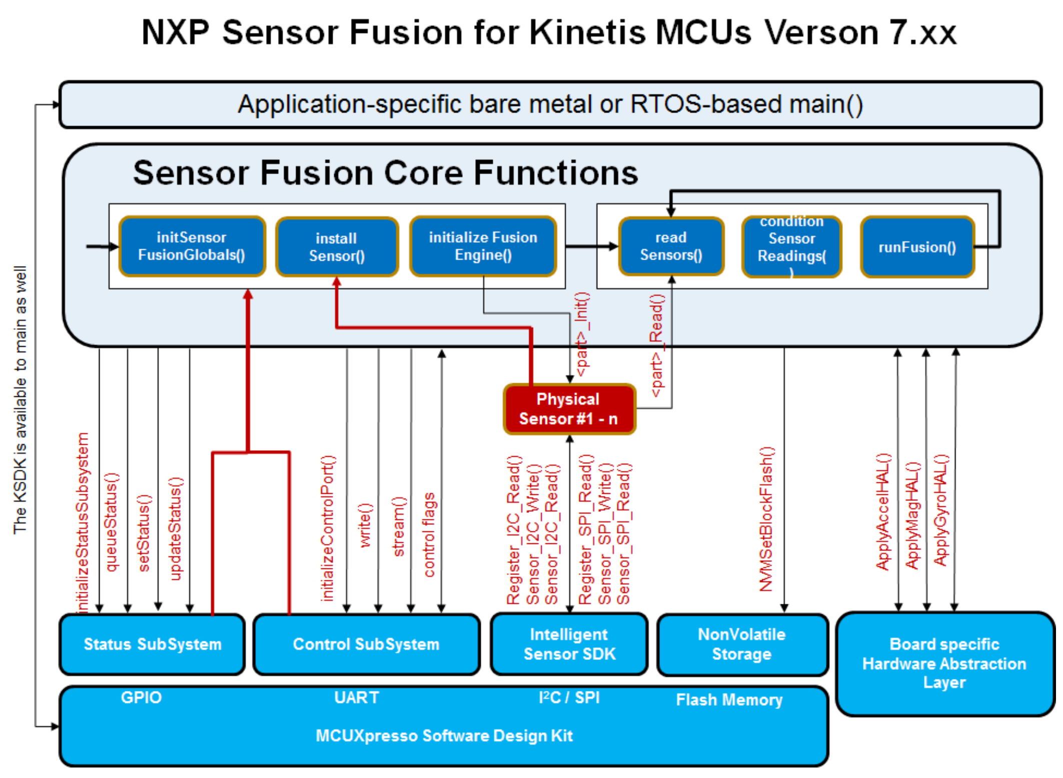


Fig 26. High Level Architecture

The board-specific Hardware Abstraction Layer (HAL) does not require the MCUXPRESSO SDK. It is responsible for mathematically aligning the axes of all sensors used to compute orientation.

The core functions of the sensor fusion library are designed for use with or without an RTOS. The status and control subsystems, which are considered design dependent, are dynamically installed at run time. In a similar fashion, the **installSensor()** function is used to specify which sensors are to be used to provide raw data for the fusion process. Each sensor driver must include a function to initialize the sensor, and another to read values.

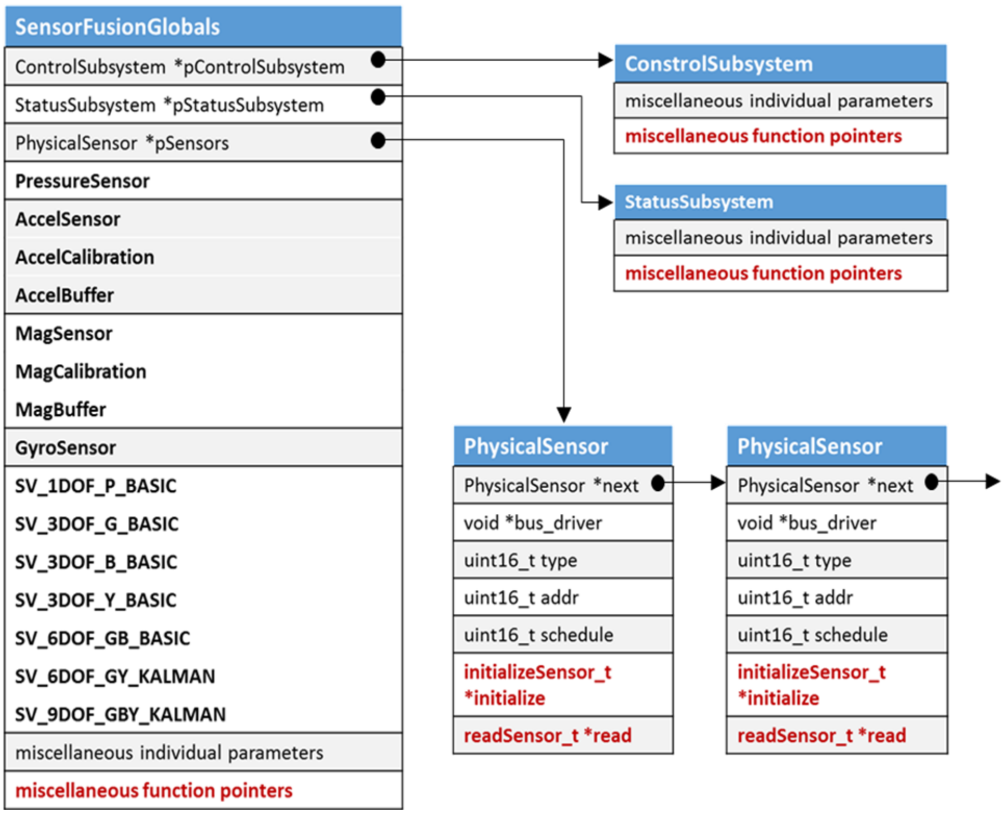
The **initializeFusionEngine()** sets up all the fusion engine data structures. Then **readSensors(), conditionSensorReadings()** and **runFusion()** functions are called periodically to read and process new data. For bare-metal implementations, loop timing can be provided by any hardware timer. A periodic interval timer, or PIT, driver is provided as an example. Various Real Time Operating systems will generally have their own mechanisms. Often this may utilize the ARM SysTick timer.

4.2 Data structures

C header files have been extensively annotated with for use with Doxygen. The development kit will ship with a software reference manual in HTML format. Details of individual data structures can be found there.

4.2.1 High level view

The V7.xx design kit never calls **malloc()**. All storage is allocated at compile time and defined statically in the calling program. To make this easier, this version of the kit has in-lined structures which, in previous releases, were freestanding.



(1) Bold black are in-line structures

(2) Bold red are function pointers

(3) Normal text = scalars and pointers

Fig 27. Sensor Fusion high level data structures

The top level structure is of type **SensorFusionGlobals**. As seen in Fig 27, everything associated with a sensor fusion application can be accessed starting with an instance of this structure. The structure grows and shrinks as a function of compile-time configuration parameters defined in build.h.

By convention, we declare this structure:

```
SensorFusionGlobals sfg;
```

in our main.c. The address of structure sfg is commonly passed to fusion routines, which then use pointer de-referencing to access data. This user guide will use this notation in examples from this point forward. For instance:

```
B[CHX] = sfg->Mag.fBc[CHX];
```

reads the averaged, calibrated, magnetometer measurement in the X direction.

Note that an object-oriented style of programming has been used, wherein public functions associated with a given system are referenced via function pointers in the applicable structure. So the high level sensor fusion functions are accessed via the **SensorFusionGlobals** structure, the **ControlSubsystem** functions via that structure, and the same for the **StatusSubsystem**. The exception to this general rule are those functions necessary to populate the structures in the first place.

4.2.2 Sensor data structures

The sensor fusion development kit supports sensors of type:

- Pressure/altimeter
- Accelerometer
- Magnetometer
- Gyroscope

Each sensor type has an associated substructure defined within the **SensorFusionGlobals** structure. It is the responsibility of the sensor **<sensor>_Read()**[4] functions to populate raw sensor readings into these structures at the start of each loop through the sensor fusion.

In the case of pressure/altimeter sensors, this is a simple write operation directly into the appropriate fields (i.e., **iH** and **iT**) of the structure.

The other three sensor types utilize software FIFOs, which can be filled incrementally over the course of several sampling loops prior to processing by the fusion routines. In these cases, you use the **addToFifo()** function to transfer values from the sensor's hardware FIFO into the fusion library's software FIFO. A good example of how to do this is contained within driver_MMA8652.c.

Compile time build.h parameters which control which sensor structures are used are:

```
#define F_USING_ACCEL        0x0001
#define F_USING_MAG   0x0002
#define F_USING_GYRO  0x0004
#define F_USING_PRESSURE     0x0008
#define F_USING_TEMPERATURE  0x0010
```

If the sensor type is needed, use the values shown. Otherwise change the unneeded field value to 0x000.

4.2.3 Reading sensor values

Note: *CHX, CHY and CHZ are conveniently defined as 0, 1 and 2 for readability in the library. Sensor data types are defined in sensor_fusion.h.*

All of the examples that follow assume that **sfg** is a pointer to the top level sensor fusion structure.

4.2.3.1 Accelerometer

Accelerometer values are stored as int16_t. Multiply by sfg->Accel->fgPerCount to convert to gravities. Both calibrated and uncalibrated results are available.

For calibrated outputs:

```
void getAccel1(SensorFusionGlobals *sfg, float fAcc[3])
{
```

4. [4] defined within the corresponding driver_<sensor>.c file.

```
  fAcc[CHX] = (float) sfg->Accel.iGc[CHX]*sfg->Accel.fgPerCount;
  fAcc[CHY] = (float) sfg->Accel.iGc[CHY]*sfg->Accel.fgPerCount;
  fAcc[CHZ] = (float) sfg->Accel.iGc[CHZ]*sfg->Accel.fgPerCount;
}
```

Alternately, simply read the averaged measurements, which are already stored as floats:

```
void getAccel2(SensorFusionGlobals *sfg, float fAcc[3])
{
  fAcc[CHX] = sfg->Accel.fGc [CHX];
  fAcc[CHY] = sfg->Accel.fGc [CHY];
  fAcc[CHZ] = sfg->Accel.fGc [CHZ];
}
```

for uncalibrated, use iGs and fGs instead of iGc and fGc above. Please note that calibrated results differ from uncalibrated results only if you are using the precision accelerometer calibration functions of the toolkit.

4.2.3.2 Magnetometer

Magnetometer values are stored as int16_t. Multiply by thisMag->fuTPeruCount to convert to μTs. Both raw and calibrated magnetometer outputs are available. Scaling is the same for both.

For calibrated outputs:

```
void getMag1(SensorFusionGlobals *sfg, float fMag[3])
{
  fMag[CHX] = (float) sfg->Mag.iBc[CHX]*sfg->Mag.fuTPerCount;
  fMag[CHY] = (float) sfg->Mag.iBc[CHY]*sfg->Mag.fuTPerCount;
  fMag[CHZ] = (float) sfg->Mag.iBc[CHZ]*sfg->Mag.fuTPerCount;
}
```

OR

```
void getMag2(SensorFusionGlobals *sfg, float fMag[3])
{
  fMag[CHX] = sfg->Mag.fBc [CHX];
  fMag[CHY] = sfg->Mag.fBc [CHY];
  fMag[CHZ] = sfg->Mag.fBc [CHZ];
}
```

For uncalibrated outputs, use iBs and fBs instead of iBc and fBc in the code above.

4.2.3.3 Gyroscope

Gyroscope values are stored as int16_t. Multiply by thisGyro->fDegPerSecPerCount to convert to degrees/second. Uncalibrated readings can be read:

```
void getAV1(SensorFusionGlobals *sfg, float fAV[3])
{
  fAV[CHX] = (float) sfg->Gyro.iYs[CHX]* sfg->Gyro.fDegPerSecPerCount;
  fAV[CHY] = (float) sfg->Gyro.iYs[CHY]* sfg->Gyro.fDegPerSecPerCount;
  fAV[CHZ] = (float) sfg->Gyro.iYs[CHZ]* sfg->Gyro.fDegPerSecPerCount;
}
```

OR

```
void getAV2(SensorFusionGlobals *sfg, float fAV[3])
{
  fAV[CHX] = sfg->Gyro.fYs [CHX];
  fAV[CHY] = sfg->Gyro.fYs [CHY];
  fAV[CHZ] = sfg->Gyro.fYs [CHZ];
}
```

Calibrated gyro measurements are available as angular velocity numbers computed by the Kalman filters in the sensor fusion library. They are read as the **iOmega** fields in the state vector structures associated with the two different Kalman filter options shown in Table 6.

4.2.3.4 Pressure sensor / altimeter

The sensor fusion library includes a driver for the MPL3115A2 pressure/sensor altimeter. This driver configures the 3115 to output an estimate of altitude, which is based upon the fixed NASA standard atmospheric model [10]. Altitude estimates will vary as a function of pressure and temperature, and are best used as a relative indicator of altitude change.

These can be retrieved in a fashion similar to the sensor types already described.

```
void checkAltimeter1( SensorFusionGlobals *sfg,
                      float *altitude,
                      float *temperature)
{
  *altitude = (float) sfg->Pressure.iH*sfg->Pressure.fmPerCount;
  *temperature = (float) sfg->Pressure.iT*
     sfg->Pressure.fCPerCount;
}
```

OR

```
void checkAltimeter2( SensorFusionGlobals *sfg,
                      float *altitude,
                      float *temperature)
{
  *altitude = sfg->Pressure.fH;
  *temperature = sfg->Pressure.fT;
}
```

4.2.4 State vector structures for fusion algorithms

Each fusion algorithm has its own dedicated state vector (SV_) data structure within the larger **SensorFusionGlobals**. Each has a compile time control similar to those discussed in the previous section within build.h. Table 5 enumerates the options.

Table 5. State Vector Substructures within SensorFusionGlobals
Algorithm names match those used by the Sensor Fusion Toolbox for Windows

Sub-Structure	Algorithm	build.h control
SV_1DOF_P_BASIC	Altimetry	F_1DOF_P_BASIC
SV_3DOF_G_BASIC	Tilt	F_3DOF_G_BASIC
SV_3DOF_B_BASIC	2D Automotive Compass	F_3DOF_B_BASIC
SV_3DOF_Y_BASIC	Rotation	F_3DOF_Y_BASIC
SV_6DOF_GB_BASIC	Tilt Compensated	F_6DOF_GB_BASIC

Sub-Structure	Algorithm	build.h control
	eCompass	
SV_6DOF_GY_KALMAN	Gaming Handset	F_6DOF_GY_KALMAN
SV_9DOF_GBY_KALMAN	Gyro Stabilized Compass	F_9DOF_GBY_KALMAN

The routines which compute orientation (all but the first in the table above) return a common set of parameters (such as output quaternion), as well as algorithm-specific values in their output data structures. Table 6 lists the C field names within each of those structures. Because the field names are determined based on algorithm-specific naming conventions, they do not always match for equivalent fields. The **SV_COMMON** and **SV_ptr** data types can be used to cast any of the six structure types into a common naming convention.

For instance, the following utility function is used to scale streaming fusion results for transmission over a serial line to the Sensor Fusion Toolbox:

```
void readCommon( SV_ptr data,
                 Quaternion *fq,
                 int16_t *iPhi,
                 int16_t *iThe,
                 int16_t *iRho,
                 int16_t iOmega[],
                 uint16_t *isystick)
{
     *fq = data->fq;
     iOmega[CHX] = (int16_t) (data->fOmega[CHX] * 20.0F);
     iOmega[CHY] = (int16_t) (data->fOmega[CHY] * 20.0F);
     iOmega[CHZ] = (int16_t) (data->fOmega[CHZ] * 20.0F);
     *iPhi = (int16_t) (10.0F * data->fPhi);
     *iThe = (int16_t) (10.0F * data->fThe);
     *iRho = (int16_t) (10.0F * data->fRho);
     *isystick = (uint16_t) (data->systick / 20);
}
```

readCommon() could then be called:

```
SV_ptr A;
A = ((SV_ptr)&sfg->SV_3DOF_G_BASIC; // or
A = ((SV_ptr)&sfg->SV_3DOF_B_BASIC; // or
A = ((SV_ptr)&sfg->SV_3DOF_Y_BASIC; // or
A = ((SV_ptr)&sfg->SV_6DOF_GB_BASIC; // or
A = ((SV_ptr)&sfg->SV_6DOF_GY_KALMAN; // or
A = ((SV_ptr)&sfg->SV_9DOF_GBY_KALMAN;
readCommon(A, &fq, &iPhi, &iThe, &iRho, iOmega, &isystick);
```

Table 6 defines the values found in the various state vector structures, along with the generalized name defined in the **SV_COMMON** type definition.

Table 6. Location of individual variables within the global structures

Description	Data Type	Fusion algorithm option						SV_COMMON
		G (accel)	B (auto compass)	Y (gyro)	GB (eCompass)	GY (accel + gyro)	GBY (9-axis)	
roll in degrees	float	fLPPhi	fLPPhi	fPhi	fLPPhi	fPhiPl	fPhiPl	fPhi
pitch in degrees	float	fLPThe	fLPThe	fThe	fLPThe	fThePl	fThePl	fThe
yaw in degrees	float	fLPPsi	fLPPsi	fPsi	fLPPsi	fPsiPl	fPsiPl	fPsi
compass heading in degrees	float	fLPRho	fLPRho	fRho	fLPRho	fRhoPl	fRhoPl	fRho
tilt angle in degrees	float	fLPChi	fLPChi	fChi	fLPChi	fChiPl	fChiPl	fChi
orientation matrix (unitless)	float	fLPR[3][3]	fLPR[3][3]	fR[3][3]	fLPR[3][3]	fRPl[3][3]	fRPl[3][3]	fRM[3][3]
quaternion (unitless)	Quaternion	fLPq	fLPq	fq	fLPq	fqPl	fqPl	fq
rotation vector	float	fLPRVec[3]	fLPRVec[3]	fRVec	fLPRVec[3]	fRVecPl[3]	fRVecPl[3]	fRVec[3]
angular velocity in dps	float	fOmega[3]	fOmega[3]	fOmega[3]	fOmega[3]	fOmega[3]	fOmega[3]	fOmega[3]
SysTick	Int32_t	SysTick	SysTick	SysTick	SysTick	SysTick	SysTick	SysTick
magnetic inclination angle in degrees	float	N/A	N/A	N/A	fLPDelta	N/A	fDeltaPl	

Description	Data Type	Fusion algorithm option						SV_COMMON
		G (accel)	B (auto compass)	Y (gyro)	GB (eCompass)	GY (accel + gyro)	GBY (9-axis)	
gyro offset in degrees/sec	float	N/A	N/A	N/A	N/A	fbPL[3]	fbPL[3]	
linear acceleration in the global frame in gravities	float	N/A	N/A	N/A	N/A	fAccGl[3]	fAccGl[3]	
time interval in seconds	float	fdeltat	fdeltat	fdeltat	fdeltat	deltat	deltat	

Se

4.2.5 Secret decoder ring for variable names

You may wonder where the various variable names in the earlier sections come from. This section answers that question.

Fusion Algorithm Options correspond to the following:

G	= SV_3DOF_G_BASIC (accel only)
B	= SV_3DOF_B_BASIC (2-axis auto compass)
Y	= SV_3DOF_B_BASIC (gyro integration)
GB	= SV_6DOF_GB_BASIC (accel + mag eCompass)
GY	= accel + gyro Kalman
GBY	= 9-axis Kalman

Table 6 variable names follow a strict naming convention.

- Core variable names
    - m = magnetic
    - b = gyro offset
    - a = acceleration
    - q = quaternion
- angle names= Phi, The, Psi, Rho and Chi
- f prefix = floating point variable
- LP = low pass filtered
- Pl suffix = a posteriori estimate from Kalman filter
- Gl suffix = global frame
- R = rotation / orientation
- Se = sensor frame

4.3 Status subsystem

The status subsystem provides a visual indication of the current state of the sensor fusion system. The default implementation is to simply control the color and rate of the tricolor LEDs available on many NXP Freedom boards.

This subsystem includes the following source files:

- status.c (core functions)
- status.h (primary interface)
- sensor_fusion.c (encapsulates core functions)
- sensor_fusion.h (defines **fusion_status_t** and function typedefs)

The only consumer of the system status is the status subsystem itself. Consider it a “write only memory”. You should feel free to swap out the default implementation with your own so long as it adheres to the same interface – even if it has zero functionality.

Status states which can be set are of type fusion_status_t. Legal values are shown in Table 7.

Table 7. Legal status states

State Name	LED Status	Description
OFF	OFF	The application hasn't started
INITIALIZING	Solid Green	Initializing sensors and algorithms
NORMAL	Blinking Green	Operation is Nominal
HARD_FAULT	Solid Red	Nonrecoverable FAULT = something went very wrong.
SOFT_FAULT	Red Flash	Recoverable FAULT = something went wrong, but we can keep going

Functions for setting status are encapsulated and inherited by the core fusion system. As seen by your application, they are defined in Table 8.

Table 8. Functional interface for the status subsystem
sss = pointer to status subsystem, sfg=pointer to sensor fusion globals, State = variable of type fusion_status_t.

Command	Function
initializeStatusSubsystem(sss)	Initializes hardware interfaces and data structures
sfg->setStatus(sfg, State)	Changes system status immediately
sfg->queueStatus(sfg, State)	Queue up a **next status** which will take effect the next time that updateStatus is called.
sfg->updateStatus(sfg)	Promotes previously queued **next status** to current status.

The low level functions for implementing this subsystem are status.c and status.h.

Note that when you set your status to HARD_FAULT, the application will halt in the status function. The intent is that HARD_FAULT conditions should never occur in normal practice. So never returning from the set/update status function call makes it easier to examine the call stack in your debugger.

Install your status subsystem using the **initSensorFusionGlobals()** core function at the start of your **main()**.

4.4 Control subsystem

The control subsystem is responsible for all serial communication between the MCU running sensor fusion and either version of the Sensor Fusion Toolbox. Fusion controls maintained by this subsystem are shown in Table 9.

Table 9. Controls maintained by the control subsystem

Field	Used by	Function
DefaultQuaternionPacketType	sensor_fusion.c	default quaternion transmitted at power on
QuaternionPacketType	Initial value set in sensor_fusion.c	quaternion type transmitted over UART
AngularVelocityPacketOn	Only the control subsystem itself	flag to enable angular velocity packet
DebugPacketOn		flag to enable debug packet
RPCPacketOn		flag to enable roll, pitch, compass packet
AltPacketOn		flag to enable altitude packet
AccelCalPacketOn	sensor_fusion.c	variable used to coordinate accelerometer calibration

This subsystem includes the following source files:

- control.c (core functions)
- control.h (primary interface)
- DecodeCommandBytes.c (command decoder)
- output_stream.c (creates streams to Sensor Fusion Toolbox)

Table 10. Functional interface for the control subsystem
sfg = pointer to sensor fusion globals, sBuffer is a large (256) character buffer.

Command	Function
initializeControlPort(&controlSubsystem)	Initializes hardware and software interfaces for the control subsystem.
sfg.pControlSubsystem->stream(&sfg, sBuffer);	Stream sensor and fusion data to the Sensor Fusion Toolbox

It is possible to build an application that does not include a control subsystem, but it is highly recommended that this capability be kept at least through initial PCB and algorithm debug. As mentioned elsewhere, NXP Community support generally requires that developers have access to the features of the Sensor Fusion Toolbox for Windows, which requires this interface for proper operation.

The packet protocol for this system is defined in Section 7.

Peripheral definitions may vary from chip to chip. Filename control.c is the “standard” control subsystem implementation in the kit. Alternate implementations are: control_lpsci.c (used for the FRDM-KL25Z implementation) and control_lpc.c (used for the MCUXpresso54114 implementation).

Install your control subsystem using the **initSensorFusionGlobals()** core function at the start of your **main()**. See Section 4.6.1 for additional details.

4.5 Sensor drivers

NXP supplies a number of sensor drivers with the Sensor Fusion Library. If you choose to add your own, they should adhere to the same design philosophy and structure outlined in the following subsections.

If you are using NXP-supplied drivers on supported NXP boards, you may skip this section.

4.5.1 Driver philosophy

Let us define two types of sensors:

- A logical sensor of one specific physical quantity. Examples include acceleration, magnetic field, angular velocity, etc.
- A physical sensor that contains 1 or more logical sensors. This is typically called a **combo sensor**.

A sensor driver must be supplied for each physical sensor. That driver is responsible for handling all of the logical sensors contained in that device package. An example of a physical device containing more than one logical sensor type is the NXP FXOS8700 Accel/Mag combo sensor.

Each sensor driver contains two to three functions:

1. The first, <sensor>_Init(), is responsible for initializing all logical sensors to start sampling at regular intervals. This function is required.

2. The second, <sensor>_Read(), is responsible for reading sensor values and transferring them into the appropriate fields within the **SensorFusionGlobals** structure. This function is required.
3. The third, <sensor>_Idle(), can be optionally used to power down sensors when not needed. Generally, you won't need these for accelerometers as these are the lowest power of the three motion sensor types. We likely will want to keep accelerometers active to detect motion, whereas gyroscopes and magnetometers can sometimes be placed in a lower power state.

The only parameter typically affecting sensor initialization is desired sampling rate. These are set as a function of build.h options at compile time.

We use a polling methodology to simplify software structure. This means there may be some variable latency from sample to sample, but this is typically many times less than the physical bandwidth of the motion being measured, and can usually be ignored.

4.5.2 Foundation functions

Fig 26 shows the interaction between the sensor drivers (labeled Physical Sensor #1-n) and the rest of the system. Drivers are extremely simple and each requires only three underlying functions. For I^{2}C-based devices, these are:

- **Register_I2C_Read()**
- **Sensor_I2C_Write()**
- **Sensor_I2C_Read()**

For SPI-based devices, these are:

- **Register_SPI_Read()**
- **Sensor_SPI_Write()**
- **Sensor_SPI_Read()**

The Register_xxx_Read() commands read a specified number of bytes starting at a given address offset into a user-provided buffer.

The Sensor_xxx_Read() and Sensor_xxx_Write() are slightly higher level functions. They access sensor registers that have been specified using a structure of type registerreadlist_t or registerwritelist_t respectively.

If you are porting the sensor fusion library to an alternate hardware or software environment, you can reuse the existing drivers simply by swapping in your own implementation of the read and write functions above.

These functions specify register details using two different typdefs:

```
typedef struct {
     uint16_t writeTo;          ← Address
     uint8_t value;             ← Value
     uint8_t mask;              ← Generally un-used Mask. Set to 0x00;
} registerwritelist_t;
```

and

```
typedef struct {
     uint16_t readFrom;         ← Address where the value is read
     uint8_t numBytes;          ← Number of bytes to read
}    registerreadlist_t;
```

These are used to create arrays of registers to be read/written by the functions above.

The following code segments show the register definitions used by the MAG3110 read and write functions, which will be examined in the sections that follow.

```
const registerreadlist_t MAG3110_WHO_AM_I_READ[] =
{
    { .readFrom = MAG3110_WHO_AM_I, .numBytes = 1 },
    __END_READ_DATA__
};

const registerreadlist_t MAG3110_DR_STATUS_READ[] =
{
    { .readFrom = MAG3110_DR_STATUS, .numBytes = 1 },
    __END_READ_DATA__
};

registerreadlist_t MAG3110_DATA_READ[] =
{
    { .readFrom = MAG3110_OUT_X_MSB, .numBytes = 6 },
    __END_READ_DATA__
};

const registerwritelist_t MAG3110_Initialization[] =
{
     { MAG3110_CTRL_REG1, 0x00, 0x00 },
     { MAG3110_CTRL_REG2, 0x90, 0x00 },
#if (MAG_ODR_HZ <= 10) // select 10Hz ODR
     { MAG3110_CTRL_REG1, 0x19, 0x00 },
#elif (MAG_ODR_HZ <= 30) // select 20Hz ODR
     { MAG3110_CTRL_REG1, 0x11, 0x00 },
#elif (MAG_ODR_HZ <= 40) // select 40Hz ODR
     { MAG3110_CTRL_REG1, 0x09, 0x00 },
#else // select 80Hz ODR
     { MAG3110_CTRL_REG1, 0x01, 0x00 },
#endif
     __END_WRITE_DATA__
};
```

Notice how the last write in the initialization sequence is dependent upon the MAG_ODR_HZ parameter defined in build.h. Also note that each array sequence is terminated with __END_WRITE_DATA__. This is a required stopping point utilized by lower level functions that process these arrays.

4.5.3 <sensor>_Init (required)

The standard typdef for sensor initialization functions is:

```
typedef int8_t (initializeSensor_t) (
    struct PhysicalSensor *sensor,
    struct SensorFusionGlobals *sfg
) ;
```

The **sensor** parameter is a pointer to a structure of type **PhysicalSensor**, which you declare statically in your main.c. A pointer to it is passed to the **installSensor()** function at startup. The second parameter is simply a pointer to the **SensorFusionGlobals** structure.

This prototype MUST be used for all sensor initialization functions.

The function that **<sensor>_Init()** functions must perform are:

- check the sensor whoAmI to confirm that expected hardware is present
- configure the sensor for periodic sampling
- store any required constants to convert values read from registers into useful values

Let us examine the **MAG3110_Init()** function. First, we have the function declaration consistent with initializeSensor_t shown above.

```
int8_t MAG3110_Init(  PhysicalSensor *sensor,
                      SensorFusionGlobals *sfg)
{
```

define a couple local variables

```
     int32_t status;
     uint8_t reg;
```

Now call the **Register_I2C_Read()** foundation function to confirm the hardware is present and check the whoAmI.

```
     status = Register_I2C_Read(
         sensor->bus_driver,
         &sensor->deviceInfo,
         sensor->addr,
         MAG3110_WHO_AM_I, 1, &reg);
```

Return early if an error is found

```
     if ( status==SENSOR_ERROR_NONE) {
          sfg->Mag.iWhoAmI = reg;
          if (reg!=MAG3110_WHOAMI_VALUE) {
                return(SENSOR_ERROR_INIT);
          }
     } else {
          return(SENSOR_ERROR_INIT);
     }
```

Perform the set of register writes defined earlier to put the part in the proper mode.

```
     status = Sensor_I2C_Write(
            sensor->bus_driver,
            &sensor->deviceInfo,
            sensor->addr,
            MAG3110_Initialization
     );
```

Stash some constants for later use.

```
     sfg->Mag.iCountsPeruT = MAG3110_COUNTSPERUT;
     sfg->Mag.fCountsPeruT = (float)MAG3110_COUNTSPERUT;
     sfg->Mag.fuTPerCount = 1.0F / MAG3110_COUNTSPERUT;
```

Set some flags used elsewhere for power management.

```
    sensor->isInitialized = F_USING_MAG;
    sfg->Mag.isEnabled = true;
```

Return the status code received when we did configuration.

```
    return (status);
}
```

Combo devices may be more complex, but the basic flow is the same for all sensor initialization functions.

4.5.4 <sensor>_Read (required)

Read functions use essentially the identical sensor prototype as the initialization functions:

```
typedef int8_t (readSensor_t) (
    struct PhysicalSensor *sensor,
    struct SensorFusionGlobals *sfg
) ;
```

The parameters are the same previously discussed.

The function of the **<sensor>_Read** functions is straight forward, read all available values from a physical sensor, adding those into the appropriate software FIFO within the sensor fusion structures. Devices with no hardware FIFO will have only 1 value to transfer. Devices with hardware FIFOs may have more. The MAG3110 we are using as an example does not have a hardware FIFO.

```
int8_t MAG3110_Read(
    PhysicalSensor *sensor,
    SensorFusionGlobals *sfg)
{
```

You will need to declare a buffer large enough to receive the largest possible sensor payload.

```
    uint8_t I2C_Buffer[6];     ← I2C read buffer
    int8_t status;             ← I2C transaction status
    int16_t sample[3];         ← Reconstructed sample
```

Here is the actual read.

```
    status = Sensor_I2C_Read(
        sensor->bus_driver,
        &sensor->deviceInfo,
        sensor->addr,
        MAG3110_DATA_READ,
        I2C_Buffer
    );
```

Convert bytes read into the proper int16_t form. This may vary from sensor to sensor.

```
    sample[CHX] = (I2C_Buffer[0] << 8) | I2C_Buffer[1];
    sample[CHY] = (I2C_Buffer[2] << 8) | I2C_Buffer[3];
    sample[CHZ] = (I2C_Buffer[4] << 8) | I2C_Buffer[5];
```

If no errors have been found, let’s do some basic conditioning and transfer the new value into the software FIFO. For FIFO-equipped sensors, this will require a loop.

```
    if (status==SENSOR_ERROR_NONE) {
            conditionSample(sample); // truncate neg values to -32767
            sample[CHZ] = -sample[CHZ]; // +Z should point up
            addToFifo(
                (FifoSensor*) &(sfg->Mag),
                MAG_FIFO_SIZE, sample);
        } and return the status from our reads.
        return (status);
    }
```

As you can see, the driver functions are intended to be very simple, with little or no "knobs to twist". Their job is to set the sensor up once, and then transfer readings into the software FIFOs whenever called by the program scheduler. The latter is either a simple loop in **main()** for bare metal, or an RTOS ReadSensor task.

Note: *the PressureSensor structure type does not support a software FIFO. This is because the relatively low data rates associated with this type of sensor simply don't require one.*

4.5.5 <sensor>_Idle (optional)

The <sensor>_Idle() functions use the same function prototype as the initialization and read functions. These functions must perform the following operations:

- Place the sensor into a low-power mode (hopefully with a minimum wakeup latency)
- Clear the sensor->isInitialized flag
- Clear the appropriate sfg-><type>.isEnabled flag, where <type> is one of: Pressure, Accel, Mag or Gyro

When a sensor has been idled, the fusion routines can continue using the last computed reading for the given sensor. Calling <sensor>_Init() again should return the sensor to the active state.

See Section 4.8 for a description of how to implement fusion standby mode in your application.

4.5.6 Scheduling sensor read operations

Most sensors have the ability to schedule periodic samples based upon their own internal timer. Each sensor may either raise an interrupt to the MCU to read its values, or simply wait until the MCU polls the sensor. Either way, the clock used by the sensor will most likely not be in perfect synchronization with the MCU clock. When you consider that systems like ours have multiple sensors, it gets more complicated.

This is important, because the equations of motion at the heart of this and many other fusion-based systems assume that all quantities are measured at the same moments in time. This may not even be possible to do, as there is no common set of sampling frequencies that apply to all sensors.

The easiest way to deal with this problem is to ignore it via the simple expedient of polling all sensors at a fixed rate, while setting their internal sample rates high enough that you can ignore any sample time jitter. You might even want to sample at a much higher rate so that you can filter raw sensor samples to reduce noise. This approach has been taken by all prior versions of the sensor fusion library.

Version 7.xx of the fusion library adds support for sensor hardware FIFOs. So we end up with a lot of variables to consider:

1. the physical bandwidth of the motion we are measuring
2. the fusion rate
3. sensor noise levels
4. the software sampling rate
5. hardware sampling rates for
    - accelerometer
    - magnetometer
    - gyroscope
6. FIFO sizes for the sensors above
7. system power requirements

Many basic human motions have a fairly low bandwidth of 12 Hz or less[1]. However athletes may exceed this many times over. Consider a baseball player swinging a bat for instance. You will typically want to set your fusion rate at least twice that of the motion you wish to capture.

Given a desired fusion rate, how fast do you need to sample your sensors? Generally as fast as you can within your power budget. The noisiest sensor in the system tends to be the magnetometer. It is subject to both internal sensor noise, as well as spurious fields within the environment. This noise can have visible effects on orientation results from eCompass and 9-axis fusion algorithms. Oversampling and averaging sensor samples helps to minimize those effects.[5]

In Version 5.00 and earlier of the fusion library, all sensors were sampled at the same sampling rate, and hardware FIFOs in the sensors were not utilized. Although workable, it did limit the ability to utilize each sensor at its optimal settings.

Addition of hardware FIFO support fixes that problem. As long as you read sensor values fast enough that FIFOs do not fill, you can decouple the various sensor sampling rates.

The software sampling rate should be an integer multiple of the fusion rate. Generally, you want the software sampling rate to be at least as high as the sampling rate for the slowest sensor in your system that is not equipped with a hardware FIFO. For the NXP sensor set supported by the standard kit, this will be the MAG3110, which has a maximum rate of 80Hz.

For a MAG3110-equipped system running a fusion rate at 25Hz, you probably want your software sampling rate at 100Hz. Your gyro and accelerometer can be programmed for higher rates, and because they are equipped with hardware FIFOs, you can read them at the same, or even a lower rate than the MAG3110!

A key parameter passed to the system each time a sensor driver is the **schedule** parameter. Set it to **1** to read a particular sensor each time the **readSensors()** function is called. Set it to **2** for every other time, **4** for every fourth time, etc. readSensors divides the **loop_counter** by schedule, if the remainder is zero, the sensor is sampled. The loop_counter variable itself is controlled by the calling program.

[5] Assuming you have zero-mean evenly distributed noise, you can add *n* effective bits of resolution to a signal by oversampling and averaging at a rate of 2^{2n}. So oversampling by 4X gives you an extra bit of resolution, 16X gives you two bits.

4.6 Core functions

Core Functions refers to the top level function calls used to integrate sensor fusion into an application. By design these are few and simple. They hide a tremendous amount of complexity below the surface.

- **initSensorFusionGlobals()**
- **installSensor()**
- **initializeFusionEngine()**
- **readSensors()**
- **conditionSensorReadings()**
- **runFusion()**
- **applyPerturbation()** (debug only)

4.6.1 InitSensorFusionGlobals

Variable space for sensor fusion is defined statically above main():

```
SensorFusionGlobals sfg;
ControlSubsystem controlSubsystem
StatusSubsystem statusSubsystem
```

All of these must be initialized before use:

```
initializeControlPort(&controlSubsystem);
initializeStatusSubsystem(&statusSubsystem);
initSensorFusionGlobals(  &sfg,
                          &statusSubsystem,
                          &controlSubsystem
);
```

At approximately 9K bytes for the full fusion demo, the **sfg** instance of **SensorFusionGlobals** is by far the largest consumer of RAM in the system, as it encompasses most of the structures used by the sensor fusion system.

The control and status subsystems are both designed with functions responsible for setting up required hardware and software interfaces. Once these are called, you must call **the initSensorFusionGlobals()** function to install the subsystems for sensor fusion use and initialize internal variables and software interfaces.

4.6.2 InstallSensor

Fusion applications vary in terms of what sensors are used and how often they are sampled. A way was needed to inform the fusion engine of new sensor types. This is done with the **installSensor()** function.

First, you must define storage for a **PhysicalSensor** for each sensor component used in your application. This should be dropped into your main.c just below the statements covered in the prior section:

`PhysicalSensor sensors[3];` ← This implementation uses three physical sensors

Then you call **installSensor**() once per each physical sensor. For example:

```
sfg.installSensor(
     &sfg,            ← Pointer to the SensorFusionGlobals structure
     &sensors[0]      ← storage for this physical sensor
```

```
    0x1E,              ← I2C address of the sensor (zero for SPI sensors)
    1,                 ← sample schedule parameter
    (void*) I2Cdrv,    ← SDK serial port driver to drive communications
    registerDeviceInfo_t *busInfo,
                       ← Specify background task to run during serial accesses
    FXOS8700_Init,     ← The initialization function from the sensor driver
    FXOS8700_Read      ← The read function from the sensor driver
);
```

See Section 4.5.5 for a description of how to use the **sample schedule parameter**.

4.6.3 initializeFusionEngine

inializeFusionEngine() is easy to use. It is typically the last fusion function you call before you begin the fusion and sampling loops. It is called with one parameterthat is a pointer to your **SensorFusionGlobals** structure.

```
sfg.initializeFusionEngine(&sfg);
```

initializeFusionEngine() performs the following functions:

- Enables the ARM SysTick counter
- Sets the initial status using the status control subsystem
- Calls the **<sensor>_Init** function for all installed sensors
- Determines the default quaternion type to be communicated to the Sensor Fusion Toolbox
- Initializes all applicable sensor fusion algorithms
- Initializes the magnetic calibration routines
- If applicable, preloads accelerometer calibration parameters
- Clears software FIFOs

4.6.4 readSensors

The **readSensors()** function simply loops through all installed sensor types. If the (read_loop_counter/sensor schedule parameter) is zero, then sample the sensor.

4.6.5 conditionSensorReadings

conditionSensorReadings() calls three lower-level functions to process accelerometer, magnetometer and gyroscope data. The general flow for these is:

- Apply hardware abstraction layer to mathematically align each sensor reading to the same desired frame of reference
- Issue a soft fault error if the software FIFO was overrun
- Compute an average of the integer results (mag and accel)
- Integrate gyro rotations
- Compute floating point averages
- For accelerometer and magnetometer
  - Compute calibrated readings
  - Update calibration system data buffers

4.6.6 runFusion

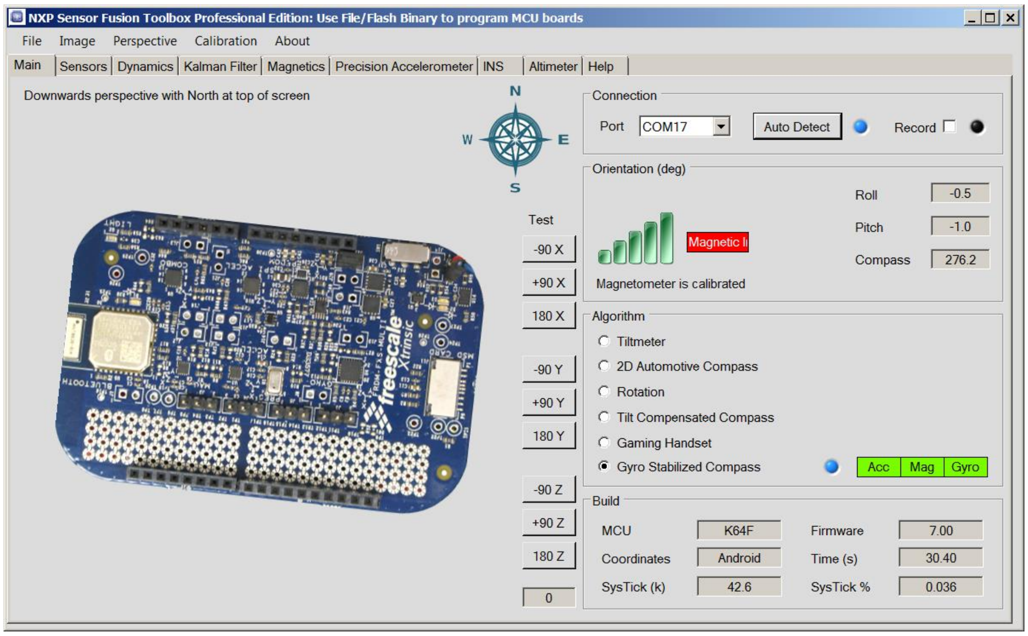
runFusion() does two things:

- Run the fusion routine(s)
- Clear software FIFOs

Angular velocity vectors out of the fusion routines have the same units as gyroscope readings, but they have been corrected to account for gyro offsets and other errors.

4.6.7 applyPerturbation (optional)

The **applyPerturbation()** function is only useful when you have the default status subsystem installed and your board is controlled via the Sensor Fusion Toolbox for Windows as shown in Fig 28.



NXP Sensor Fusion Toolbox Professional Edition: Use File/Flash Binary to program MCU boards

Downwards perspective with North at top of screen

Connection: Port COM17, Record ☐

Orientation (deg): Roll -0.5, Pitch -1.0, Compass 276.2

Magnetometer is calibrated

Algorithm: ( ) Tiltmeter, ( ) 2D Automotive Compass, ( ) Rotation, ( ) Tilt Compensated Compass, ( ) Gaming Handset, [✓ Gyro Stabilized Compass]

Build			
MCU	K64F	Firmware	7.00
Coordinates	Android	Time (s)	30.40
SysTick (k)	42.6	SysTick %	0.036

Fig 28. Step function test features in the Sensor Fusion Toolbox for Windows

In the center of the display, to the right of the floating PCB, you will see a number of buttons labeled -90X through 180Z. If you click one of these buttons, the toolbox will fool the fusion routines on the board into thinking that the prior orientation was off by this amount. If things are working correctly, you should see the floating PCB flip to that orientation and then almost immediately rotate back as the fusion routines correct for the perceived error.

4.7 RTOS or not?

Version 5.00 and earlier versions of the sensor fusion library were tightly integrated into the MQX RTOS. This made it difficult to port them to another RTOS or use in a bare-metal implementation. That all changed with Version 7.xx. The RTOS has been decoupled, and exactly the same functions can be used in both environments.

Using an RTOS allows you to easily juggle more activities at a given time, but the approach does have a cost in terms of complexity and code size.

Bare-metal implementations collapse the sensor sampling and fusion loops into one loop, controlled by a hardware timer. Hardware FIFO support mentioned earlier means that, even with this simplistic approach, you can often achieve reasonable performance.

The code for both implementations is simple enough to be included in their entirety in the following sections. We have removed comments from the original and annotated the code as has been done in prior sections. It might be a good idea to open the original commented files in an editor to follow along with the discussions that follow.

4.7.1 Bare metal main

Filename main_baremetal.c provides the source code for the remainder of this section.

4.7.1.1 Required #includes

The first set of headers are part of the MCUXPRESSO SDK installation for your MCU.

```
// MCUXPRESSO SDK Headers
#include "fsl_debug_console.h
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_port.h"
#include "fsl_i2c.h"
#include "fsl_smc.h"
#include "register_io_i2c.h"
#include "fsl_i2c_cmsis.h"
```

The second set of header files are required for the sensor fusion functions. The mixture of <sensor>.h files will change as a function of your particular hardware.

Notice the inclusion of driver_pit.h. We will be using a periodic interval timer (PIT) as the source of our main loop timing. You can use any available timer.

```
// Sensor Fusion Headers
#include "fsl_pit.h"
#include "fxas21002.h"
#include "mpl3115.h"
#include "fxos8700.h”

#include "sensor_fusion.h"
#include "control.h"
#include "status.h”
#include "drivers.h"
#include "driver_pit.h"
```

There are two important files that are not shown. Both are pulled in by sensor_fusion.h

- issdk_hal.h pulls in Hardware Abstraction Layer functions for your particular board(s) and may (along with lower level HAL files) need to be customized if using your own PCB design.
- build.h specifies build parameters for the sensor fusion library.

4.7.1.2 Global storage

KDSK has introduced new CMSIS drivers for I^{2}C and SPI. These allow the application to place the MCU in a low power mode while waiting for serial transfers to complete.

The user can specify a function pointer and function argument by setting the idleFunction and functionParam structure entries in the structure below. This is then associated with the sensor driver via the installSensor() function.

The 1st version shown here is equivalent to past implementations, in that the MCU does not enter into sleep mode.

```
registerDeviceInfo_t i2cBusInfo = {
    .deviceInstance     = I2C_S_DEVICE_INDEX,
    .functionParam      = NULL,
    .idleFunction       = NULL
};
```

Alternately:

```
registerDeviceInfo_t i2cBusInfo = {
    .deviceInstance     = I2C_S_DEVICE_INDEX,
    .functionParam      = SMC,
    .idleFunction = (registeridlefunction_t) SMC_SetPowerModeWait
};
```

will place the MCU in a low power state during transfers. You will need a 2nd instance of the structure above If your design utilizes a SPI port.

The next set of statements have already been discussed in Sections 4.2, 4.2.5, 4.4 and 4.6.2.

```
SensorFusionGlobals sfg;
ControlSubsystem controlSubsystem;
StatusSubsystem statusSubsystem;
PhysicalSensor sensors[2];
```

4.7.1.3 bare metal main()

And now to the heart of things. The first couple blocks in main do nothing more than standard variable declaration and hardware setup. Note that I2C_S_DRIVER_BLOCKING is defined as a function of the issdk_hal.h contents.

```
int main(void)
{
    ARM_DRIVER_I2C* I2Cdrv = & I2C_S_DRIVER_BLOCKING;
    uint16_t i=0;
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();
```

Here we initialize the I2C port.

```
    I2Cdrv->Initialize(I2C_S_SIGNAL_EVENT);
    I2Cdrv->PowerControl(ARM_POWER_FULL);
    I2Cdrv->Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_FAST);
```

Now we start initializing the various sensor fusion systems. These statements will apply to almost every application.

```
     initializeControlPort(&controlSubsystem);
     initializeStatusSubsystem(&statusSubsystem);
     initSensorFusionGlobals(
         &sfg,
         &statusSubsystem,
         &controlSubsystem
     );
```

This particular build only requires two sensors, located at I²c addresses 0x1E and 0x20. The **installSensor()** function is discussed in detail in Section 4.6.2

```
    sfg.installSensor(
            &sfg,                  ← Pointer to the global structure
            &sensors[0],           ← Pointer to PhysicalSensor structure
            0x1E,                  ← Sensor I²C address
            1,                     ← Sample everytime readSensors() is called
            (void*) I2Cdrv,        ← Serial port driver
            &i2cBusInfo,           ← Serial port driver power control
            FXOS8700_Init,         ← Sensor driver initialization function
            FXOS8700_Read          ← Sensor driver read function
    );
    sfg.installSensor(
            &sfg,
            &sensors[1],
            0x20,
            1,
            (void*) I2Cdrv,
            &i2cBusInfo,
            FXAS21002_Init,
            FXAS21002_Read
    );
```

Initialize the fusion engine. This was discussed in detail in Section 4.6.3.

```
    sfg.initializeFusionEngine(&sfg);
```

We have written a simple PIT driver for use in timing the main loop. Here we call its initialization function and set it to a period defined via the FUSION_HZ constant defined in build.h. Then set our system status to NORMAL.

```
    pit_init(1000000/FUSION_HZ);
    sfg.setStatus(&sfg, NORMAL);
```

And finally, here is our main loop. **pitIsrFlag** is set in the PIT interrupt service routine and cleared here.

```
    while (true)
    {
            if (true == pitIsrFlag) {
            sfg.readSensors(&sfg, 1);
            sfg.conditionSensorReadings(&sfg);
            sfg.runFusion(&sfg);
            sfg.applyPerturbation(&sfg);
            sfg.loopcounter++;
            i=i+1;
            if (i>=4) {
                    i=0;
                    sfg.updateStatus(&sfg);
            }
```

```
                    sfg.queueStatus(&sfg, NORMAL);
                    sfg.pControlSubsystem->stream(
                         &sfg,
                         sUARTOutputBuffer
                    );
                    pitIsrFlag = false;
               }
          }
}
```

Notice that we do not stream data back to the Sensor Fusion Toolbox until the end of each loop. Everything is serialized and happens exactly in the sequence shown.

4.7.2 FreeRTOS main.c

Filename main_freertos_two_tasks.c provides the source code for the remainder of this section. We have applied the same treatment as the prior section, but have used color coding to indicate where the two implementations differ.

4.7.2.1 Required #includes

We've had to add a number of FreeRTOS includes over those needed for the bare-metal implementation.

```
/* FreeRTOS kernel includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "event_groups.h"
```

The next set of headers are part of the MCUXPRESSO SDK installation for your MCU. They are the same as the bare-metal version, with the subtraction of fsl_pit.h. FreeRTOS uses the ARM SysTick counter as its default timebase, therefore we no longer need the PIT.

```
// MCUXPRESSO SDK Headers
#include "fsl_debug_console.h
#include "board.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "fsl_port.h"
#include "fsl_i2c.h"
#include "register_io_i2c.h"
#include "fsl_smc.h"
```

The next set of includes are required for the sensor fusion functions. Notice the exclusion of driver_pit.h.

```
// Sensor Fusion Headers
#include "fxas21002.h"
#include "mpl3115.h"
#include "fxos8700.h"
#include "sensor_fusion.h"
#include "control.h"
#include "status.h"
```

```
#include "drivers.h"
```

As noted in the earlier section, there are two important files that are not shown. Both are pulled in by sensor_fusion.h

- issdk_hal.h pulls in Hardware Abstraction Layer functions for your particular board(s) and may (along with lower level HAL files) need to be customized.
- buid.h specifies build parameters for the sensor fusion library.

4.7.2.2 Global storage

The first set of statements have already been discussed in Sections 4.2, 4.2.5, 4.4 and 4.6.2. We are adding an altimeter (the NXP MPL3115A2) to this build, so the sensors array is larger by one than the bare-metal implementation. The **event_group** will be utilized later to coordinate between FreeRTOS tasks.

```
SensorFusionGlobals sfg;
ControlSubsystem controlSubsystem;
StatusSubsystem statusSubsystem;
PhysicalSensor sensors[3];
EventGroupHandle_t event_group = NULL;

registerDeviceInfo_t i2cBusInfo = {
    .deviceInstance     = I2C_S_DEVICE_INDEX,
    .functionParam      = SMC,
    .idleFunction       = (registeridlefunction_t)
                              SMC_SetPowerModeWait
};
```

4.7.2.3 FreeRTOS specifics

You need to declare the real-time tasks that the RTOS will be executing. We have two for this build. The **fusion_task()** will run at FUSION_Hz and the read_task() will run at FAST_LOOP_HZ. Both constants are defined in build.h. FAST_LOOP_HZ should also be used to set the **configTICK_RATE_HZ** parameter defined in **FreeRTOSConfig.h**. A FreeRTOS configuration file is expected to be present in every FreeRTOS project.

```
static void read_task(void *pvParameters
static void fusion_task(void *pvParameters
```

You should review the FreeRTOS documentation to get a better handle on FreeRTOSConfig.h contents. In particular, the **configMINIMAL_STACK_SIZE** and **configTOTAL_HEAP_SIZE** parameters can easily make your application nonfunctional.

4.7.2.4 A simpler main()

main() no longer has a loop of any kind, so we have eliminated our counter variable.

```
int  main(void)
{
     ARM_DRIVER_I2C* I2Cdrv = &I2C_S_DRIVER_BLOCKING
     BOARD_InitPins();
     BOARD_BootClockRUN()
     BOARD_InitDebugConsole();
```

I^{2}C initialization is identical to the bare-metal case.

```
     I2Cdrv->Initialize(I2C_S_SIGNAL_EVENT);
     I2Cdrv->PowerControl(ARM_POWER_FULL);
```

```
    I2Cdrv->Control (
        ARM_I2C_BUS_SPEED,
        ARM_I2C_BUS_SPEED_FAST
    );
```

I2C_S_DRIVER_BLOCKING has been defined in a <shield>.h file to match up with the I2C port that communicates with the sensors in question. The <shield>.h file is pulled in via issdk_hal.h.

The fusion systems are initialized identically to the bare-metal case.

```
    initializeControlPort(&controlSubsystem);
    initializeStatusSubsystem(&statusSubsystem);
    initSensorFusionGlobals(
        &sfg, &statusSubsystem,
        &controlSubsystem
    );
```

Our first two sensors are identical to the bare-metal case. Addition of the altimeter here is independent of the choice of RTOS or not.

```
sfg.installSensor(
     &sfg, &sensors[0],
     0x1E,
     1,
     (void*) I2Cdrv,
     &i2cBusInfo,
     FXOS8700_Init,
     FXOS8700_Read
);
sfg.installSensor(
     &sfg,
     &sensors[1],
     0x20,
     1,
     (void*) I2Cdrv,
     &i2cBusInfo,
     FXAS21002_Init,
     FXAS21002_Read
);
sfg.installSensor(
     &sfg,
     &sensors[2],
     0x60,
     2,
     (void*) I2Cdrv,
     &i2cBusInfo,
     MPL3115_Init,
     MPL3115_Read
);
sfg.initializeFusionEngine(&sfg);
```

We have eliminated the PIT in favor of creating FreeRTOS controls.

```
    event_group = xEventGroupCreate();
    xTaskCreate(
            read_task,
            "READ",
            configMINIMAL_STACK_SIZE,
            NULL,
            tskIDLE_PRIORITY + 2,
            NULL
 );
 xTaskCreate(
            fusion_task,
            "FUSION",
            configMINIMAL_STACK_SIZE,
            NULL,
            tskIDLE_PRIORITY + 1,
            NULL
 );
```

We still need to set the initial status. But then we start the FreeRTOS scheduler, which should never return. If it does, we flag a HARD_FAULT.

```
    sfg.setStatus(&sfg, NORMAL);
    vTaskStartScheduler();
    sfg.setStatus(&sfg, HARD_FAULT);
    for (;;) ;
}
```

Note: *if vTaskStartScheduler() returns, it most likely is because you did not allocate enough memory in your FreeRTOSConfig.h file.*

4.7.2.5 read_task

The read_task() operates at FAST_LOOP_HZ in this implementation.

```
static void read_task(void *pvParameters)
{
    uint16_t i=0;
    portTickType lastWakeTime;
    const portTickType frequency = 1;
    lastWakeTime = xTaskGetTickCount();
    while (1)
    {
```

We have a short loop nested within an infinite loop. The **OVERSAMPLE_RATE**, which is defined in build.h, determines the ratio of sensor cycles to fusion cycles. **vTaskDelayUntil()** is a FreeRTOS function that blocks operation for a specified interval. **xEventGroupSetBits()** sets an event bit to trigger the fusion task. We are taking advantage of the loop interval to determine how often to sample our various sensors within the **readSensors()** function.

```
    for (i=1; i<=OVERSAMPLE_RATE; i++) {
            vTaskDelayUntil(&lastWakeTime, frequency);
            sfg.readSensors(&sfg, i);
    }
    xEventGroupSetBits(event_group, B0);
  }
}
```

4.7.2.6 fusion_task

The **fusion_task()** runs at FUSION_HZ for this implementation. Notice that the **xEventGroupWaitBits()** blocking FreeRTOS command is the primary addition to this function. It blocks execution until the B0 **event_group** bit was set in the **read_task()**. Otherwise, all the components here can also be found in our bare-metal implementation.

```
static void fusion_task(void *pvParameters)
{
     uint16_t i=0; // general counter variable
     while (1)
     {
          xEventGroupWaitBits( event_group,
                               B0,
                               pdTRUE,
                               pdFALSE,
                               portMAX_DELAY);

          sfg.conditionSensorReadings(&sfg);
          sfg.runFusion(&sfg);
          sfg.applyPerturbation(&sfg);

          sfg.loopcounter++;
          i=i+1;
          if (i>=4) {
                    i=0;
                    sfg.updateStatus(&sfg);
          }
          sfg.queueStatus(&sfg, NORMAL);
          sfg.pControlSubsystem->stream(&sfg, sUARTOutputBuffer);
     }
}
```

4.7.3 Key takeaway

The key message to take away from the prior sections is that the sensor fusion library is now RTOS-agnostic. The same functions can be used in an RTOS environment or in a bare-metal implementation. We have used FreeRTOS to illustrate the point, however there is no reason to suppose that the library could not be used with *any* RTOS.

See “Converting a FreeRTOS Project to Bare Metal” for an explicit recipe for converting FreeRTOS projects back to bare metal.

4.8 Fusion Standby mode

In Sections 4.5.1 and 4.5.5, we introduced the <sensor>_Idle() function built into some sensor drivers. It can be used to place affected sensors in a low-power state until the <sensor>_Init() function is called again. This gives us the basis for implementing a mode of operation in which the application can reduce power during periods of inactivity.

4.8.1 motionCheck

The motionCheck() function is not a sensor fusion function. It is a function that simply monitors an accelerometer or magnetometer triaxial sensor output, returning Boolean true if the sensor appears to be stationary and false otherwise. The function prototype is:

```
bool motionCheck( float sample[3],
                  float baseline[3],
                  float tolerance,
                  uint32_t winLength,
                  uint32_t *count)
```

parameters are:

- **sample** is a 3-element array containing the processed sensor sample. For accelerometers, these are in units of gravity (*g*s).
- **baseline** is a 3-element array containing the value to which sample is being compared. If the two arrays do not match within the specified tolerance (the 3rd parameter), then baseline is updated to the current sample value.
- **tolerance** is the amount you have defined to specify the threshold for what you consider a change. The value of **tolerance** must have the same units as sample and baseline. It also must be above the "noise threshold" of the sensor.
- **winLength** is the number of fusion cycles in which you must see no motion before asserting the "no motion" flag.
- **count** is the number of cycles in which no change has been detected. This tops out at (winLength + 1) to prevent counter overflow.

4.8.2 Sample implementation

Standby mode is implemented at the application level within the fusion task or loop, not within the fusion library itself. The basic idea is to power down gyro (and possibly magnetometer) when the board is not moving, and to even skip the fusion call itself. Since the board is stationary, there is no need to re-compute orientation on every pass through the loop!

Let us explore the changes required to the FreeRTOS fusion_task to implement this functionality. Changes are shown in **red**.

First, note that we must provide the variables required for the checkMotion() function.

```
static void fusion_task(void *pvParameters)
{
    uint16_t i=0; // general counter variable
    float motion_baseline[3] = {0.0, 0.0, 0.0};
    bool stationary;
    static bool lastStationary;
    uint32_t stationaryCount = 0;
    while (1)
    {
        xEventGroupWaitBits( event_group,
                             B0,
                             pdTRUE,
                             pdFALSE,
                             portMAX_DELAY);

        sfg.conditionSensorReadings(&sfg);
```

It is important that the conditionSensorReadings() call happen before the motionCheck() call, as you would like to make this check using the cleanest version of the signal you have available.

```
        stationary =  motionCheck(
                      sfg.Accel.fGc, // calibrated accel reading
                      motion_baseline,
                      0.01,
                      120,
                      &stationaryCount);
```

Here is where we handle the transition between modes. Notice that the runFusion() call only occurs when the device is in motion.

```
        if (stationary) {
            if (!lastStationary) { // go into standby
                FXAS21002_Idle(&(sensors[1]), &sfg);
            }
            clearFIFOs(&sfg);
        } else {
            if (lastStationary) { // restart operations
                FXAS21002_Init(&(sensors[1]), &sfg);
            }
            sfg.runFusion(&sfg); // fuse the sensor data
        }

        sfg.loopcounter++;
        i=i+1;
        if (i>=4) {
                i=0;
                sfg.updateStatus(&sfg);
        }
```

Change the LED indicator depending upon our current “motion status”.

```
        if (stationary) {
            sfg.queueStatus(&sfg, LOWPOWER);
        } else {
            sfg.queueStatus(&sfg, NORMAL);
        }
        sfg.pControlSubsystem->stream(
            &sfg,
            sUARTOutputBuffer);
        lastStationary = stationary;
     }
}
```

Then, using a tolerance of 10 m*g* (0.01 g), the transition between modes is imperceptible when viewing the motion of the board via the Sensor Fusion Toolbox.

The code shown above does not deal at all with MCU low-power modes. These are outside the scope of the sensor fusion library.

4.9 Reading results from global structures

Section 4.2.3 presented example code segments for reading average sensor outputs. Raw sensor readings may be read in a similar manner. And accelerometer and magnetometer outputs are available in both calibrated and uncalibrated forms. Consult the reference manual pages or source code for sensor_fusion.h for further details.

Section 4.2.4 presents similar information for accessing:

ϕ = fPhi = roll angle in degrees

θ = fThe = pitch angle in degrees

ψ = fPsi = yaw angles in degrees

ρ = fRho = compass heading in degrees

χ = fChi = tilt angle from vertical in degrees

fRM[3][3] = orientation matrix

fq = orientation quaternion

fRVec[3] = rotation vector

ω = fOmega[3] = average angular velocity in degrees/sec

SysTick = the number of system clocks required to run the algorithm

On an algorithm-by-algorithm basis, other variables may be available. Consult Table 6 for the location of various parameters within the various structures.

With the exception of loading raw data into the **PressureSensor** structure, the user should never write to any sensor or algorithm structure. These should be considered as "read-only".

5. Customizing your build

This chapter presents a number of recipes for common modifications to the standard sensor fusion build. Should you not find what you need in this user guide, submit a question to the NXP Sensor Fusion Community.

5.1 Selecting which algorithms to build

The algorithms in the fusion library have been heavily optimized. You can run any combination of the algorithms in the development kit on many ARM® Cortex® M0+ platforms, and almost all Cortex M4F platforms.

The process of choosing which algorithms to run is easy:

1. Change the #define for THISCOORDSYSTEM in build.h to one of NED (aka Aerospace), ANDROID or WIN8. This will define the global frame of reference you want your algorithm to work in.

   Refer to NXP Application Note AN5017, Rev 1.1: Aerospace, Android and Windows 8 Coordinate Systems for a full description of the differences between the options.

2. Modify the algorithm selection fields in build.h, see Table 11, to choose which algorithm builds to include or exclude from your build.

Table 11. Algorithm selection fields in build.h

Algorithm	build.h control	To Include	To Exclude
Altimetry	F_1DOF_P_BASIC	0x0100	0x0000
Tilt	F_3DOF_G_BASIC	0x0200	0x0000
2D Automotive	F_3DOF_B_BASIC	0x0400	0x0000

Algorithm	build.h control	To Include	To Exclude
Compass			
Rotation	F_3DOF_Y_BASIC	0x0800	0x0000
Tilt Compensated eCompass	F_6DOF_GB_BASIC	0x1000	0x0000
Gaming Handset	F_6DOF_GY_KALMAN	0x2000	0x0000
Gyro Stabilized Compass	F_9DOF_GBY_KALMAN	0x4000	0x0000

3. Modify the sensor selection fields in build.h, see Table 12, to pull in (at a minimum) the set of sensors required for your algorithm choice above.

Table 12. Sensor selection fields in build.h

Sensor Type	build.h control	To Include	To Exclude
Accelerometer	F_USING_ACCEL	0x0001	0x0000
Magnetometer	F_USING_MAG	0x0002	0x0000
Gyroscope	F_USING_GYRO	0x0004	0x0000
Pressure	F_USING_PRESSURE	0x0008	0x0000
Temperature	F_USING_TEMPERATURE	0x0010	0x0000

Unselected algorithm and sensor types are completely eliminated from the build at compile time, and consume no resources.

5.2 Sample and fusion rate topics

Section 4.5 dealt with sample and fusion rate topics in the context of sensor drivers. Be sure you read and understand that section before making changes.

From a checklist perspective, here are the things you will need to change to affect sample and fusion rates:

1. If you are using a bare-metal project, set the timer interval, which controls your main loop, equal to the software sample/fusion rate (they are equal). The example project shipped with the development kit allows you to do this by changing the value passed to **pit_init()** in **main()**.
2. If you are using FreeRTOS, set configTick_RATE_HZ in FreeRTOSConfig.h equal to your software sampling loop rate.
3. in build.h, confirm the values for:
   - ACCEL_FIFO_SIZE
   - MAG_FIFO_SIZE
   - GYRO_FIFO_SIZE
   - ACCEL_ODR_HZ
   - MAG_ODR_HZ

- GYRO_ODR_HZ
- FUSION_HZ (25Hz minimum recommended)
- FAST_LOOP_HZ

The FAST_LOOP_HZ parameter must match configTick_RATE_HZ.

The FUSION_HZ value should be defined so that OVERSAMPLE_RATE= FAST_LOOP_HZ/FUSION_HZ is always an integer.

5.2.1 Using the SysTick counter to measure compute times

5.2.1.1 The technique

Assuming an ARM M0+ or M4F implementation, key function calls within the fusion library are wrapped with calls that read the ARM MCU SysTick counter and then use the difference to compute how many CPU cycles it took to execute the given block. You can see this in function fFuseSensors() in fusion.c. An example is:

```
ARM_systick_start_ticks(&(pthisSV_6DOF_GY_KALMAN->systick));
fRun_6DOF_GY_KALMAN(
    pthisSV_6DOF_GY_KALMAN,
    pthisAccel,
    pthisGyro
);
pthisSV_6DOF_GY_KALMAN->systick =
    ARM_systick_elapsed_ticks(pthisSV_6DOF_GY_KALMAN->systick);
```

Notice that the result is stored away in the State Vector structure associated with the algorithm. This is how the Sensor Fusion Toolbox is able to display compute times for the different algorithms.

5.2.1.2 Limitations

All of the above works fine so long as the fusion routine in question is allowed to run un-interrupted by other tasks or interrupts. This will generally be true of bare-metal projects which have a single thread of operation. There may be the occasional UART interrupt associated with the control subsystem, but we assume those to be negligible.

Some real-time operating systems utilize the SysTick timer for their tick timer. This generally will imply that the SysTick timer will be reset at the beginning of every software sample period. On Cortex-M4F class devices, that is generally not a problem, because the fusion algorithms execute very fast, because of the integrated floating point unit. That may not be true for Cortex-M0+ or M3/M4 devices which have no floating point support.

When in doubt, resort to the age old method of inserting GPIO pin toggles at key points in your code and observing them on an oscilloscope.

5.3 Frames of reference and the HAL

3-axis sensors are assumed to have axes aligned to a Cartesian coordinate system that follows the Right Hand Rule (RHR). As an example, the figure below shows three sensors that might be used to implement a 9-axis system. Note that the Z-axis on the MAG3110 is inverted relative to the standard RHR conventions.

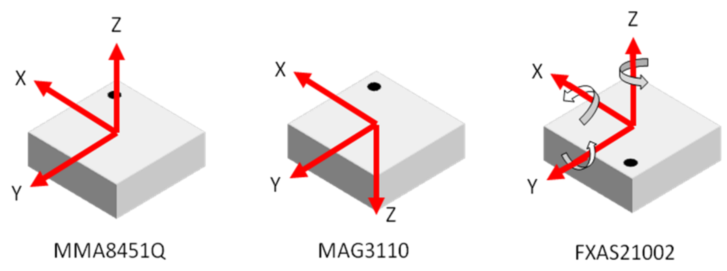


Fig 29. Problem: One of these sensors does not follow the RHR rule

That is trivial to fix. Simply invert the Z-axis readings in the MAG3110 driver. If you look at driver_MAG3110.c in the sensor fusion library, you will see that is exactly what we do:

```
sample[CHX] = (I2C_Buffer[0] << 8) | I2C_Buffer[1];
sample[CHY] = (I2C_Buffer[2] << 8) | I2C_Buffer[3];
sample[CHZ] = -((I2C_Buffer[4] << 8) | I2C_Buffer[5]);
```

The corrected axis is shown as Z' in the figure below.

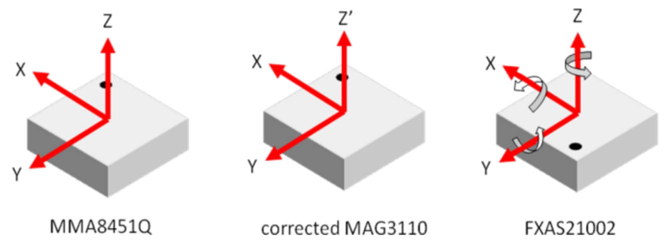


Fig 30. The Fix: Inverting the MAG3110 Z-axis in the driver

The sensor fusion algorithms also assume that your zero rotation orientation is aligned with a global RHR reference frame. Unfortunately, there are several global reference frames in use. The library natively supports Android and Windows 8 coordinate systems. These are both are East North Up (ENU) variants. It also supports the standard Aerospace North East Down (NED) coordinate system. Differences are outlined in the table below.

Table 13. Coordinate system comparison

From Application Note AN5017: Aerospace, Android and Windows 8 Coordinate Systems

Item	Aerospace (NED)	Android (ENU)	Windows 8 (ENU)
Axes alignment	XYZ = NED	XYZ = ENU	XYN = ENU
Accelerometer sign	Gravity-Acceleration +1*g* when axis is down –1*g* when axis is up	Acceleration-Gravity +1*g* when axis is up –1*g* when axis is down	Gravity-Acceleration +1*g* when axis is down –1*g* when axis is up
Accelerometer reading when flat	G[Z] = +1*g*	G[Z] = +1*g*	G[Z] = –1*g*
Direction of increasing angles	Clockwise	Counterclockwise	Clockwise
Compass heading	$\rho = \psi$	$\rho = \psi$	$\rho = 360°-\psi$

Item	Aerospace (NED)	Android (ENU)	Windows 8 (ENU)
ρ and yaw angle ψ			

So what do you do if your PCB design does not follow the conventions of your standard frame of reference? Maybe one sensor is rotated 90 degrees relative to the other two in a 9-axis system. Or maybe one of the sensors is mounted on the bottom of the PCB and the other two are mounted on the top. This is where the Hardware Abstraction Layer, or HAL comes into play. The HAL allows you to mathematically map each set of sensor readings into a single, consistent, frame of reference. NXP supplied HALs also enable you to switch from one target frame of reference to another with a simple #define flag change.

In the figure below, we have physically rotated the MMA8451Q 90 degrees from the desired ENU frame of reference.

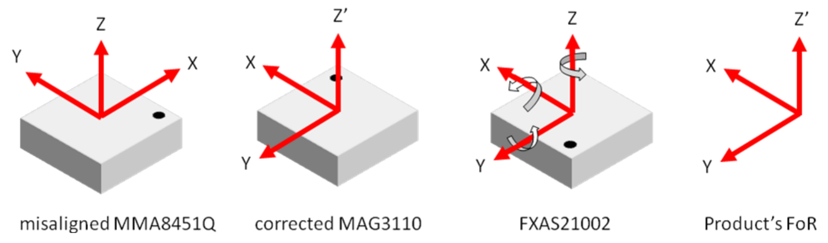


Fig 31. Problem: MMA8451 rotated 90 degrees

The process to remap the 8551Q's readings into the standard frame of reference is easy. Draw a picture of the actual and desired orientations and then write the equations of the desired sensor's readings in terms of the physical sensor.

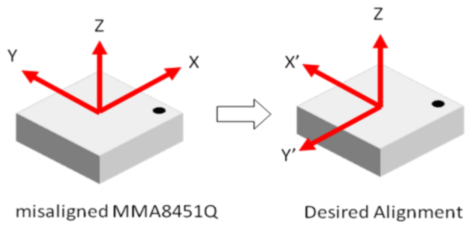


Fig 32. Draw a picture of actual versus desired reference frame

```
X'= Y;
Y'= -X;
Z = Z; // no change
```

These would then go into the HAL function which manages the accelerometer frame of reference. There are three HAL functions:

- **ApplyAccelHAL()**
- **ApplyMagHAL()**
- **ApplyGyroHAL()**

These are defined in filename hal_<boardname>.c. Standard configurations for NXP Freedom development boards are included in the kit. You may need to develop your own HAL if sensors on your board are not aligned similarly. You should start by copying one of the existing HAL files and making the minimum set of changes necessary to accommodate your sensor placement. Then modify hal.h to remove the old HAL file and add yours.

5.4 Sensor calibration topics

In Section 2.8 we discussed the need for careful sensor calibration for advanced applications like navigation. Version 5.0 and earlier of the sensor fusion library included hard/iron magnetic compensation and dynamic gyro offset computation. Version 7.xx added advanced accelerometer calibration and static gyro offset measurements. It also added the option of storing all calibration parameters in nonvolatile storage for fast startup.

5.4.1 Some background

For an at-rest system, accelerometers and magnetometers are conceptually very similar. They measure a fixed three-dimensional vector. In the case of the accelerometer, it is gravity. The magnetometer measures the earth's magnetic field. The calibration algorithms in the sensor fusion library take advantage of the fact that if you measure the same fixed 3D vector while rotating your sensor in space, and plot the points, you will ideally obtain a sphere. When you talk with NXP sensor experts, you may hear them refer to "geomagnetic sphere" or "1*g* sphere". This is what they are referring to.

If you linearly distort a sphere, you get an ellipsoid. Since the process is linear, it is possible to reverse it. That is the heart of the calibration routines for both magnetometer and accelerometer.

See *NXP Application Note AN5286, Rev. 2.0: Precision Accelerometer Calibrations* for a detailed description of the technique.

5.4.2 Accelerometer

Recall that accelerometers measure linear acceleration minus gravity. If the device is at rest, all you get is the gravity component.

NXP Application Note AN5286, Precision Accelerometer Calibration, describes the three different levels of accelerometer calibration: four parameter, seven parameter and ten parameter. The first two are proper subsets of the ten-parameter model. The choice of which to use is governed by how many measurements you are willing to take for the use as inputs to the algorithms. At the end of the day, you end up with an equation of the form:

$$ {}^{S}\boldsymbol{G}_{c,k} = (\Delta\boldsymbol{R})^{T} W^{-1}\left({}^{S}\boldsymbol{G}_{k} - \boldsymbol{V}\right) $$

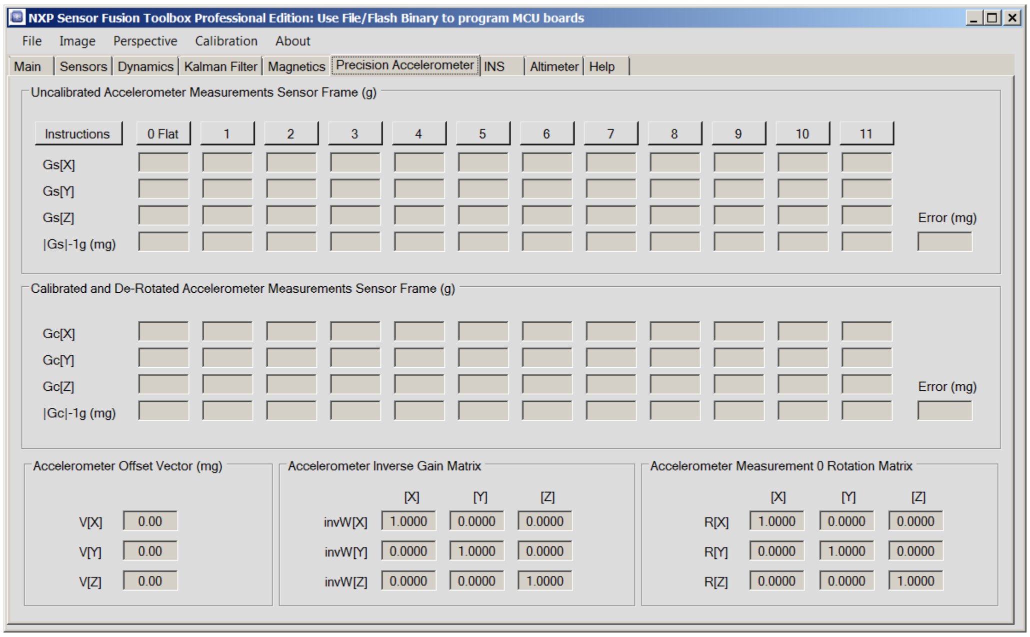
where:

${}^{S}G_{c,k}$	= the kth calibrated accelerometer reading in the sensor frame
ΔR	= tilt rotation error (a rotation matrix) of the accelerometer inside the final product
W^{-1}	= Inverse gain matrix
${}^{s}G_{k}$	=the kth uncalibrated (raw) accelerometer reading
V	= the offset vector

Note that the calibration equation above does not include environmental effects.

Computation of calibration coefficients is a manually triggered event. The user model for accelerometer calibration is to take sufficient measurements (12 recommended) to compute all parameters and store those into device NVM. Later, the final tilt rotation matrix can be adjusted using a single accelerometer reading taken on the final production line. The modified rotation matrix can then be re-stored in NVM. Thus a board could be factory programmed and then “tweaked” once installed in the final housing.

A new version of the Sensor Fusion Toolbox for Windows was released for use with the Version 7.xx Sensor Fusion Library. It has a new **Precision Accelerometer** tab that can be used to accomplish this task during development.



Uncalibrated Accelerometer Measurements Sensor Frame (g)

	0 Flat	1	2	3	4	5	6	7	8	9	10	11	Error (mg)
Gs[X]													
Gs[Y]													
Gs[Z]													
|Gs|-1g (mg)													

Calibrated and De-Rotated Accelerometer Measurements Sensor Frame (g)

													Error (mg)
Gc[X]													
Gc[Y]													
Gc[Z]													
|Gc|-1g (mg)													

Accelerometer Offset Vector (mg)

V[X]	0.00
V[Y]	0.00
V[Z]	0.00

Accelerometer Inverse Gain Matrix

	[X]	[Y]	[Z]
invW[X]	1.0000	0.0000	0.0000
invW[Y]	0.0000	1.0000	0.0000
invW[Z]	0.0000	0.0000	1.0000

Accelerometer Measurement 0 Rotation Matrix

	[X]	[Y]	[Z]
R[X]	1.0000	0.0000	0.0000
R[Y]	0.0000	1.0000	0.0000
R[Z]	0.0000	0.0000	1.0000

Fig 33. Problem: MMA8451 Rotated 90 Degrees

Clicking any of the “0 Flat”, “1” through “11” buttons instructs firmware on the board to take a series of measurements (nominally 2 second’s worth), compute the average, and then store the average into a numbered buffer for use during a least squares fit to determine V, W^{-1} and ΔR. As you add more measurements, those values will be updated using the four, seven and finally ten parameter calibration models.

The first “0 flat” measurement should be taken with the board flat on a table top. All other readings can be taken at random orientations which approximately cover the “geomagnetic sphere” discussed earlier. Fig 34 shows the jig that a member of the fusion team uses to position his board. The jig was cobbled together using a scrap piece of 2x4 lumber and some glue. There is no need for precision in building your own jig. Don’t bother asking for mechanical drawings – they do not exist.

Fig 35 shows the **Precision Accelerometer** tab after a 12-orientation calibration has been performed. The top table shows the raw measurements and error = (vector magnitude – 1*g*) computed for each. The maximum error for this particular run was over 49 m*g* on measurement #4. The value on the right side shows an average error of 25.15 m*g*. Contrast this with the table in the center of the screen. It shows a maximum calibrated error of 0.64 m*g*, with an average error of 0.34 m*g*. Over 70X improvement!

At the bottom of the screen in Fig 35, you see the computed values for V, W^{-1} and ΔR.

Bear in mind that what we have done is a linear trim based upon the vector magnitude of the averaged measurements. Impacts of sensor noise are very low because we’ve taken a lot of measurements to compute each average. Any errors left over at the end of the process are, by definition, residual nonlinearities.

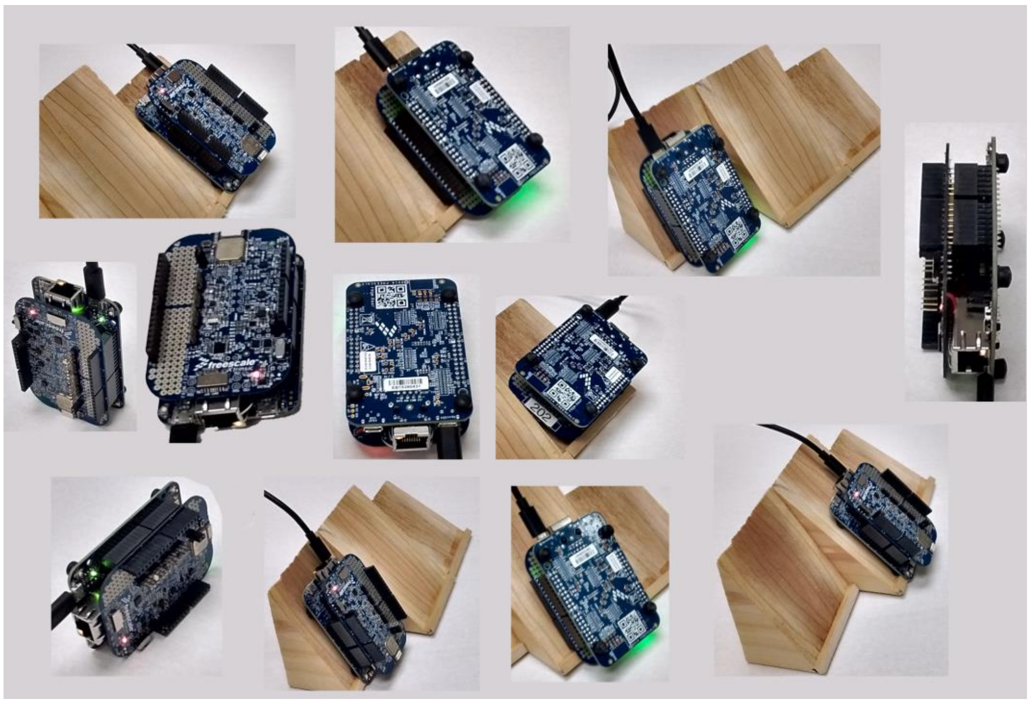
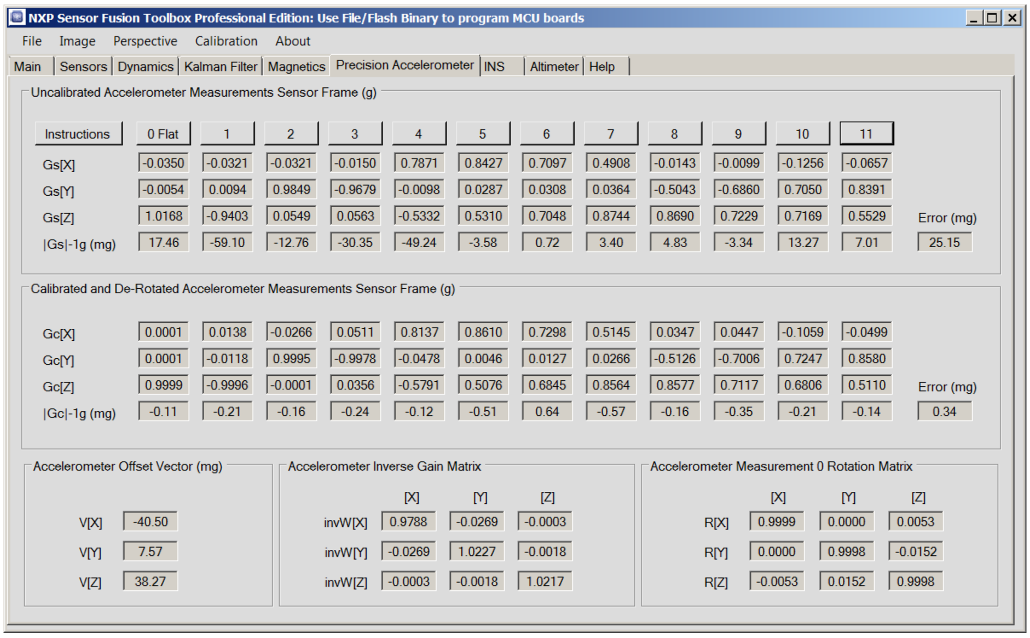


Fig 34. Precise control of measurement angles is only required once: flat



Uncalibrated Accelerometer Measurements Sensor Frame (g)

	0 Flat	1	2	3	4	5	6	7	8	9	10	11	Error (mg)
Gs[X]	-0.0350	-0.0321	-0.0321	-0.0150	0.7871	0.8427	0.7097	0.4908	-0.0143	-0.0099	-0.1256	-0.0657	
Gs[Y]	-0.0054	0.0094	0.9849	-0.9679	-0.0098	0.0287	0.0308	0.0364	-0.5043	-0.6860	0.7050	0.8391	
Gs[Z]	1.0168	-0.9403	0.0549	0.0563	-0.5332	0.5310	0.7048	0.8744	0.8690	0.7229	0.7169	0.5529	
|Gs|-1g (mg)	17.46	-59.10	-12.76	-30.35	-49.24	-3.58	0.72	3.40	4.83	-3.34	13.27	7.01	25.15

Calibrated and De-Rotated Accelerometer Measurements Sensor Frame (g)

													Error (mg)
Gc[X]	0.0001	0.0138	-0.0266	0.0511	0.8137	0.8610	0.7298	0.5145	0.0347	0.0447	-0.1059	-0.0499	
Gc[Y]	0.0001	-0.0118	0.9995	-0.9978	-0.0478	0.0046	0.0127	0.0266	-0.5126	-0.7006	0.7247	0.8580	
Gc[Z]	0.9999	-0.9996	-0.0001	0.0356	-0.5791	0.5076	0.6845	0.8564	0.8577	0.7117	0.6806	0.5110	
|Gc|-1g (mg)	-0.11	-0.21	-0.16	-0.24	-0.12	-0.51	0.64	-0.57	-0.16	-0.35	-0.21	-0.14	0.34

Accelerometer Offset Vector (mg)

V[X]	-40.50
V[Y]	7.57
V[Z]	38.27

Accelerometer Inverse Gain Matrix

	[X]	[Y]	[Z]
invW[X]	0.9788	-0.0269	-0.0003
invW[Y]	-0.0269	1.0227	-0.0018
invW[Z]	-0.0003	-0.0018	1.0217

Accelerometer Measurement 0 Rotation Matrix

	[X]	[Y]	[Z]
R[X]	0.9999	0.0000	0.0053
R[Y]	0.0000	0.9998	-0.0152
R[Z]	-0.0053	0.0152	0.9998

Fig 35. Accelerometer calibration has been completed

At this point, we have not yet stored calibration coefficients into nonvolatile storage. This will be discussed in Section 5.4.5.

5.4.3 Magnetometer

The ambient magnetic field is not nearly as well behaved as the gravity vector, especially indoors. Stray fields, ferrous materials and magnetic materials are everywhere. Inclusion of a gyroscope into 9-axis sensor fusion helps negate the effects of random variations in the environment. Section 11 discusses distortion of the magnetic field, and shows that we can mathematically eliminate hard and soft iron effects resulting from materials that are fixed spatially with respect to the sensor.

$$\boldsymbol{B_c} = \boldsymbol{W}^{-1}(\boldsymbol{B_s} - \boldsymbol{V})$$

The trim equation above is very similar to the accelerometer trim equation discussed in the previous section, and the math to determine calibration coefficients is similar as well. Application Note AN5019, Magnetic Calibration Algorithms, provides the full mathematical derivation of those equations.

In prior generations of the sensor fusion library, magnetic calibration was run continuously in a background task whenever sensor sample and fusion tasks were not running. This had the advantage of quickly determining calibration coefficients with a simple twist of the wrist while holding a sensor board.

That approach does not work well for a single-threaded bare-metal implementation of sensor fusion. The Version 7.xx sensor fusion uses the same award winning library, but it “slices” what had been the background magnetic task into smaller chucks of code. Each chunk is small enough that it can be executed serially with sensor fusion in the same thread. Table 14 lists the pros and cons of each approach.

Table 14. Magnetic Calibration Tradeoffs between V5.00 and V7.xx sensor fusion
Table description (optional)

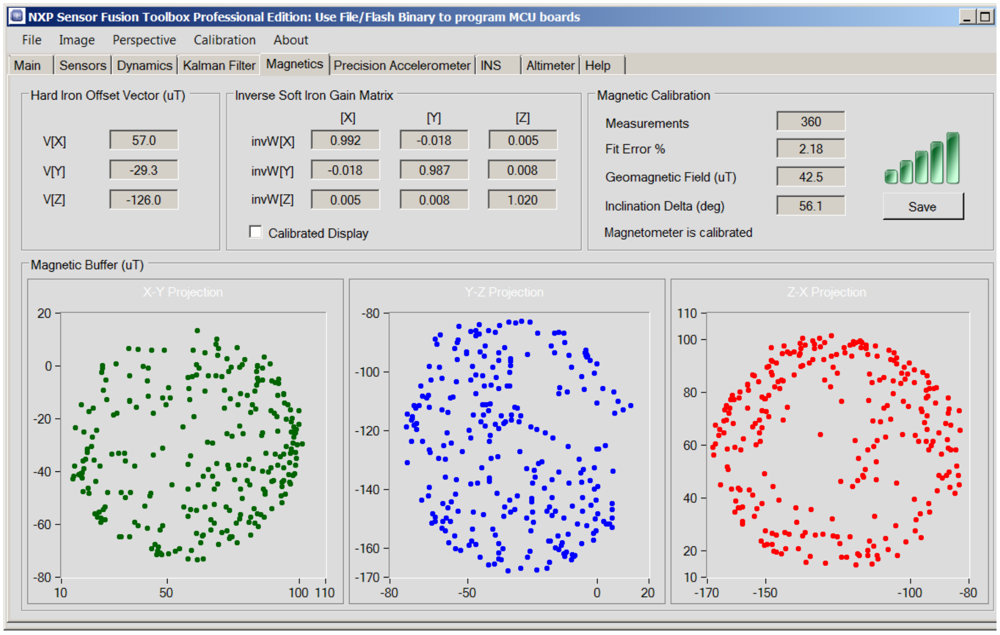
Feature	V5.00 and earlier	V7.xx
Accuracy	Same algorithms used for both	
Startup time	A few seconds	Slower if coefficients have to be computed every time. Instantaneous if NVM storage is used for pre-computed coefficients.
Implementation	Required background thread	Not even visible to most programmers

Calibrating your magnetometer with the Sensor Fusion Toolbox for Windows is easier than it was for accelerometer calibration. The fusion library automatically generates a magnetic buffer, or constellation, of data points to use for calibration. This is shown in Fig 36. Since the constellation is three dimensional, the toolbox generates plots of the data from three different perspectives. If your PCB is magnetically clean, these should be circular as shown in the figure below.

To see how this works, do the following:

1. Connect your board to the Sensor Fusion Toolbox as described in the Quick Start guide elsewhere in your manual.
2. Click the **Magnetics** tab.
3. Rotate the board a few times; you should see new points added to the constellation.

4. Click File->Reset, and the constellation will clear. If calibration data had been stored in NVM, it will be reused. Otherwise you will see an indication that the magnetometer is no longer calibrated.
5. Rotate the board again and watch as new points are added to the constellation.



Hard Iron Offset Vector (uT)

V[X]	57.0
V[Y]	-29.3
V[Z]	-126.0

Inverse Soft Iron Gain Matrix

	[X]	[Y]	[Z]
invW[X]	0.992	-0.018	0.005
invW[Y]	-0.018	0.987	0.008
invW[Z]	0.005	0.008	1.020

☐ Calibrated Display

Magnetic Calibration

Measurements	360
Fit Error %	2.18
Geomagnetic Field (uT)	42.5
Inclination Delta (deg)	56.1

Magnetometer is calibrated

Fig 36. The magnetic calibration tab

In the top row of the display, you can see the hard-iron offset vector **V** and the Inverse Soft Iron Gain Matrix $\mathbf{W}^{-1}$. Calibration points are only updated when a new calibration comes along with a better fit error as shown on the right. When starting with an uncalibrated device, the program will cycle through 4-, 7- and 10-element calibration routines as more points are added to the constellation.

5.4.4 Gyroscope

It is not uncommon for gyroscopes to have offsets, and these can drift over time. Gyroscope errors are triple integrated[6] in basic inertial navigation systems, so they can cause even larger errors than the accelerometer bias example shown in Section 2.8. The scheme used in first generation systems was to simply measure gyroscope outputs when the device is stationary. Because gyros are rate sensors, by definition the measured value equals the offset. But if your system is rarely stationary, that can be problematic. A different approach was used in Version 5.00 and earlier versions of the sensor fusion library. The gyro offsets were computed dynamically as one of the outputs of the Kalman filter. Version 7.xx of the sensor fusion library allows you to use *both* techniques.

To take a static measurement, simply place your sensor board flat on a table top prior to startup up the Sensor Fusion Toolbox for Window. The toolbox will pop a dialog box reminding you to put the unit down. Click OK and you will have your initial offset estimate.

This will be used as an initial estimate when setting up the Kalman filter.

[6] Gyroscope errors are integrated during the orientation calculation which leads to an error in the apparent gravity vector and an error in acceleration. This acceleration error is then integrated again to compute velocity and integrated once more to give displacement.

5.4.5 Storing / Retrieving Parameters from Flash Memory

Once computed, you will want to store trim parameters in nonvolatile memory so that they can be automatically loaded upon power up. During development, this is easy using the Calibration pull-down menu in the Sensor Fusion Toolbox for Windows. If you select that pull-down, you will see the following structure:

- Save to Kinetis Flash
    - Magnetic, Gyroscope and Precision Accelerometer Calibrations
    - Magnetic Calibration only
    - Gyroscope Calibration only
    - Precision Accelerometer Calibration only
- Erase from Kinetis Flash
    - Magnetic, Gyroscope and Precision Accelerometer Calibrations
    - Magnetic Calibration only
    - Gyroscope Calibration only
    - Precision Accelerometer Calibration only

These functions are implemented as part of the command protocol operating between your development board and the PC GUI. They are based on the following functions defined in calibration_storage.h:

```
void SaveMagCalibrationToNVM(SensorFusionGlobals *sfg);
void SaveGyroCalibrationToNVM(SensorFusionGlobals *sfg);
void SaveAccelCalibrationToNVM(SensorFusionGlobals *sfg);
void EraseMagCalibrationFromNVM(void);
void EraseGyroCalibrationFromNVM(void);
void EraseAccelCalibrationFromNVM(void);
```

Each of these functions is built on top of the same base function for accessing NVM memory:

```
NVM_SetBlockFlash(uint8_t *Source, uint32_t Dest, uint16_t Count)
```

where

- Source = pointer to data to be written
- Dest = pointer to destination
- Count = number of bytes to be written

The “Save” and “Erase” functions copy any existing calibration out of NVM into a buffer, modify the buffer with updated information for the appropriate sensor type and then call NVM_SetBlockFlash(), which erases and then rewrites the NVM. The implementation for NVM_SetBlockFlash() is hardware dependent. A MCUXPRESSO SDK-based implementation is provided in driver_KSDK_NVM.c/.h. If porting to a different architecture, you can preserve the same functionality simply by swapping in an equivalent version of NVM_SetBlockFlash().

It is important to understand that trim parameters stored in NVM are specific to a given instance of a PCB and Frame of Reference (Aerospace, Android or Win8). They need to be computed on a per-board basis. They also need to be recomputed if you change the frame of reference in build.h

The **trim coefficients may be lost when you reprogram the board**. This is tool chain specific. You can use the Sensor Fusion Toolbox for Windows Magnetics and Precision Acceleration tabs to determine if trim coefficients survive a re-flash event in your tool environment

5.5 Tinkering with Kalman filter parameters in fusion.h

The default filter settings in fusion.h are designed to be a “best compromise” that addresses the largest application subset possible. But invariably, questions arise, such as “How do I improve fusion tracking relative to north?”, or “How do I make my application more immune to magnetic interference?”. These two goals typically are at cross purposes. Improving one makes the other worse. You make these tradeoffs by adjusting sensor variance constants.

Parameters in fusion.h are organized by algorithm, and we follow the same organization in the subsections that follow. Each contains a table that summarizes available parameters for that algorithm.

Some general guidelines:

- Tightening sensor variances tends to put more emphasis in the Kalman filter results on values from that sensor. Loosening them has the opposite effect.
- Increasing low-pass-filter time constants will smooth results at the expense of response time.
- If there are no gyro offsets stored in NVM, the Kalman filter initialization sequences can use the first reading as the starting offsets IF it falls within the limits specified by “Permissible Power” numbers.

5.5.1 1DOF P Basic constants (pressure)

Table 15. Pressure tuning constants

Parameter	Default Value	Description
FLPFSECS_1DOF_P_BASIC	1.5	Pressure low pass filter time constant in seconds

5.5.2 3DOF G Basic constants (tilt)

Table 16. Tilt tuning constants

Parameter	Default Value	Description
FLPFSECS_3DOF_G_BASIC	1.0	Tilt orientation low pass filter time constant in seconds

5.5.3 3DOF B Basic constants (2D automotive eCompass)

Table 17. 2D Automotive eCompass Tuning Constants

Parameter	Default Value	Description

Parameter	Default Value	Description
FLPFSECS_3DOF_B_BASIC	7.0	2D eCompass orientation low pass filter time constant in seconds

5.5.4 6DOF GB Basics (eCompass)

Table 18. eCompass tuning constants

Parameter	Default Value	Description
FLPFSECS_6DOF_GB_BASIC	7.0	3D eCompass orientation low pass filter time constant in seconds

5.5.5 6DOF GY Kalman constants (gaming)

Table 19. 6-Axis Kalman Filter Tuning Constants

Parameter	Default Value	Description
FQVY_6DOF_GY_KALMAN	200	Gyro sensor noise variance in units of $(deg/s)^2$
FQVG_6DOF_GY_KALMAN	1.2E–3	Accelerometer sensor noise variance units g^2
FQWB_6DOF_GY_KALMAN	2E–2	Gyro offset random walk units $(deg/s)^2$
FMIN_6DOF_GY_BPL	–7.0	Minimum permissible power on gyro offsets (deg/s)
FMAX_6DOF_GY_BPL	+7.0	Maximum permissible power on gyro offsets (deg/s)

5.5.6 9DOF GBY Kalman constants (9-axis)

Table 20. 9-Axis Kalman Filter Tuning Constants

Parameter	Default Value	Description
FQVY_9DOF_GBY_KALMAN	200	Gyro sensor noise variance in units of $(deg/s)^2$
FQVG_9DOF_GBY_KALMAN	1.2E–3	Accelerometer sensor noise variance units g^2
FQVB_9DOF_GBY_KALMAN	5.0	Magnetometer sensor noise variance units uT^2 defining minimum deviation from geomagnetic sphere
FQWB_9DOF_GBY_KALMAN	2E–2	Gyro offset random walk units $(deg/s)^2$
FMIN_9DOF_GBY_BPL	–7.0	Minimum permissible power on gyro offsets (deg/s)
FMAX_9DOF_GBY_BPL	+7.0	Maximum permissible power on gyro offsets (deg/s)

6. Additional porting topics

6.1 Adding new sensor drivers

Section 4.5 provides a detailed review of sensor driver structure. You MUST use exactly the same prototype structure for all sensor initialization and read functions.

Generally, the easiest way to add a new driver is to copy an existing one and modify it. Once complete, add the new functions to drivers.h and use the Sensor Fusion Toolbox for Windows to verify sensor operation. See Section 9, Debugging.

6.2 Moving to different MCUs

If your new MCU of choice is a Kinetis ARM core, most of your work should be done for you, when you use MCUXpresso Config Tool to generate a SDK. If your MCU is not a Kinetis, or even an ARM core, your work is more extensive. You will need to provide drivers for:

- GPIO (used by the status subsystem)
- UART (used by the control subsystem)
- I^{2}C/SPI (used by sensor drivers)
- Nonvolatile storage (used for sensor calibration parameters)

Fig 26 illustrates the software stack-up for sensor fusion applications. The general recommendation is to study the functions used in the various control subsystems, and replicate hardware interfaces used there.

An additional item that needs to be considered is the ARM SysTick timer, which is used to measure compute times. If you do not need compute times, simply remove calls to the driver_systick.c functions. Otherwise, you will need to identify and implement an alternative that will be platform-dependent.

7. Serial packet structure

The tables presented in this section are derived from those included with the NXP Sensor Fusion Toolbox for Android in-app help.

Prior to version 2013.07.18 of the NXP Sensor Fusion Toolbox for Android application, communication between the Android device and external board was strictly one way, from board to Android device. Version 2013.07.18 and above add the ability for the application to send commands to the development board. This is done at application start-up for flags to enable the following:

- Debug mode
- Roll/pitch/compass display on the Device view
- Virtual compass display on the Device view

The debug mode is now required to be always on, because this packet transmits the firmware version number that is displayed by the Toolbox. Additionally, a packet may be sent to change quaternion type whenever the Source/Algorithm selector in the Sensor Fusion Toolbox is changed.

7.1 Development board to Fusion Toolbox

Development board to Sensor Fusion Toolbox communication uses a streaming data protocol. Packets are delimited by inserting a special byte (0x7E) between packets. This means that we must provide a means for transmitting 0x7E within the packet payload. This is done on the transmission side by making the following substitutions:

- Replace 0x7E by 0x7D5E (1 byte payload becomes 2 transmitted)
- Replace 0x7D by 0x7D5D (1 byte payload becomes 2 transmitted)

The Fusion Toolbox does the inverse mapping as the data stream is received. Partial packets are discarded. The options menu on the Android version of the toolbox has a Toggle Hex Display option available for developers coding their own board interface.

On the embedded app side, the 0x7E and 0x7D encoding is performed automatically by function `sBufAppendItem()`, which is used to create outgoing packet structures. The developer only needs to explicitly add starting and ending 0x7E delimiters.

The following packet types are supported by the protocol:

1. Orientation and Sensor Data
2. Debug
3. Angular Rate
4. Euler Angles
5. Altitude and Temperature
6. Magnetic
7. Kalman Parameters
8. Accelerometer Calibration

Packet types 9–11 are reserved.

7.1.1 Packet Type 1: Orientation and Sensor Data

This packet type is used to convey timestamp, sensor readings, orientation in quaternion form and hardware flags.

Table 21. Packet Type 1: Orientation and Sensor Data

Byte #	Function	Units
0	Packet Start = 0x7E	None
1	Packet Type = 01	None
2	Packet #	This number increments by 1 for each sample. Rolls over at 0xFF to 0x00.
6:3	Timestamp	1 LSB = 1.00 microseconds (Previously 1.33. Updated 2013.07.18)
8:7	ACC X	1 LSB = 122.07 µ*g*
10:9	ACC Y	1 LSB = 122.07 µ*g*
12:11	ACC Z	1 LSB = 122.07 µ*g*
14:13	MAGX	1 LSB = 0.1 µT
16:15	MAGY	1 LSB = 0.1 µT
18:17	MAGZ	1 LSB = 0.1 µT
20:19	GYRO X	1 LSB = 872.66 µrad/s (0.05 dps)
22:21	GYRO Y	1 LSB = 872.66 µrad/s (0.05 dps)
24:23	GYRO Z	1 LSB = 872.66 µrad/s (0.05 dps)
26:25	quaternion q0	1 LSB = 1/30,000 (unitless)

Byte #	Function	Units
28:27	quaternion q1	1 LSB = 1/30,000 (unitless)
30:29	quaternion q2	1 LSB = 1/30,000 (unitless)
32:31	quaternion q3	1 LSB = 1/30,000 (unitless)
33	Flags	Bit field with the following bit definitions: Bit 0 = valid gyro data Bit 1 = gyro is virtual Bit 2 = 6-axis quaternion is valid Bit 3 = 9-axis quaternion is valid Bits 5:4 = 00 = Data is NED (Aerospace) Frame of Reference Bits 5:4 = 01 = Data is Android Frame of Reference Bits 5:4 = 10 = Data is Windows Frame of Reference Bits 5:4 = 11 = RESERVED Bits 7:6 = RESERVED
34	Board ID	Two numeric fields with the following possible values: Base Board = Bits 4:0: 0. = RESERVED 1. = FRDM-KL25Z 2. = FRDM-K20D50M 3. = RESERVED 4. = FRDM-KL26Z 5. = FRDM-K64F 6. = RESERVED 7. = FRDM-KL46Z 8. = RESERVED 9. = FRDM-K22F 10. = RESERVED 11. = FRDM-KL05Z 12. = RESERVED 13. = FRDM-KL02Z 14. = FRDM-KE02Z 15. = FRDM-KE06Z 16. = XPRESSO_LPC5411X 17. = FRDM-KL02Z 18. = FRDM_KL05Z 19. = FRDM_KW01Z 20. = FRDM_K66Z
		Sensor Shield Board[7] = Bits 7:5 0. = FRDM-FXS-MULT2-B 1. = None 2. = FRDM-STBC-AGM01

[7] See sensor companion boards at http://www.nxp.com/freedom to determine availability.

Byte #	Function	Units
		3. = FRDM-STBC-AGM02 4. = FRDM-STBC-AGMP03 5. = FRDM-STBC-AGM04
35	Packet END = 0x7E	None

Table 21 represents the packet format adopted by the Fusion Toolbox on 1 August 2013. The assumption inherent in this format is that the application will instruct the development board with regard to desired algorithm.

Table 22 specifies the sources used to populate some of the fields above.

Table 22. Packet 1 Sources

Quantity	Derived from
Timestamp	Assumes packets are sent out at the fusion rate.
ACC	Accel.iGc
MAG	Mag.iBc
GYRO	Gyro.iYs
QUATERNION	SV_ptr->fq

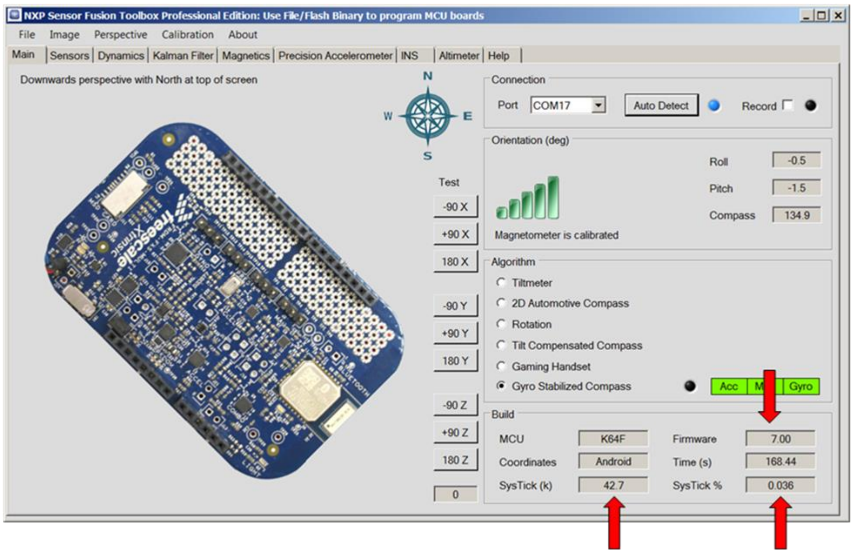
Note that Timestamps are incremented by 1,000,000/FUSION_HZ. Packet transmission is throttled to a maximum rate of 40X per second to ensure that UARTs do not exceed their maximum bandwidth.

7.1.2 Packet Type 2: Debug

The development board may send 1 to n 16-bit words to the app for display on the Device view of the Sensor Fusion Toolbox for Android. This view is enabled via a checkbox in the Preferences screen. The Windows version of the toolbox has dedicated locations in the display for software version number and SysTick numbers. It does not display optional entries.

Table 23. Packet type

Byte #	Function	Units
0	Packet Start = 0x7E	None
1	Packet Type = 0x02	
2	Packet Number	None
4:3	Firmware Version Number	None
6:5	SysTick count / 20	Bytes 6:5 now carry the SysTick count/20
…	Variable Payload	
2n + 1:2n	Debug Word*n*	Last of n debug words to transmit
2n + 2	Packet END = 0x7E	None



Downwards perspective with North at top of screen

Port: COM17

Roll: -0.5
Pitch: -1.5
Compass: 134.9

Magnetometer is calibrated

Algorithm: (●) Gyro Stabilized Compass

MCU	K64F	Firmware	7.00
Coordinates	Android	Time (s)	168.44
SysTick (k)	42.7	SysTick %	0.036

0

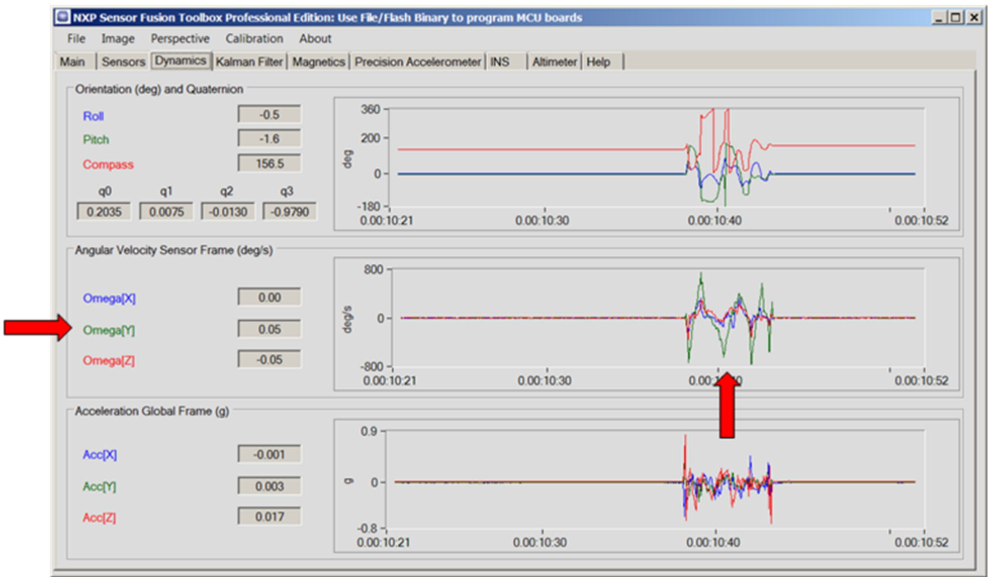
Fig 37. Debug packet information displayed on the main form

7.1.3 Packet Type 3: Angular Rate

This packet type is used to send angular rate values (**iOmega**) to the Sensor Fusion Toolbox. On the Android version, this view is enabled via a checkbox in the Preferences screen. On the Windows version, the data is shown on the **Dynamics** tab (see Fig 38).

Table 24. Packet Type 3: Angular Rate Information

Byte #	Function	Units
0	Packet Start = 0x7E	None
1	Packet Type = 0x03	None
2	Packet #	This number increments by 1 for each sample. Rolls over at 0xFF to 0x00.
6:3	Timestamp	1 LSB = 1.00 microseconds (Previously 1.33. Updated 2013.07.18)
8:7	X	1 LSB = 872.66 micro-radians/sec (0.05 dps)
10:9	Y	1 LSB = 872.66 micro-radians/sec (0.05 dps)
12:11	Z	1 LSB = 872.66 micro-radians/sec (0.05 dps)
13	Packet END = 0x7E	None


NXP Sensor Fusion Toolbox Professional Edition: Use File/Flash Binary to program MCU boards
File Image Perspective Calibration About
Main Sensors Dynamics Kalman Filter Magnetics Precision Accelerometer INS Altimeter Help
Orientation (deg) and Quaternion
deg
Angular Velocity Sensor Frame (deg/s)
deg/s
Acceleration Global Frame (g)
g

Roll -0.5
Pitch -1.6
Compass 156.5
q0 0.2035 q1 0.0075 q2 -0.0130 q3 -0.9790
Omega[X] 0.00
Omega[Y] 0.05
Omega[Z] -0.05
Acc[X] -0.001
Acc[Y] 0.003
Acc[Z] 0.017

Fig 38. Angular Velocity Display on the Sensor Fusion Toolbox for Windows

7.1.4 Packet Type 4: Euler Angles

This packet type is used to send roll/pitch/compass heading information to the Sensor Fusion Toolbox. On the Android version, this is shown on the **Device** view, and is enabled via a checkbox in the **Preferences** screen. On the Windows version, it is displayed on the **Dynamics** tab (top row of the display in Fig 38.

Byte #	Function	Units
0	Packet Start = 0x7E	None
1	Packet Type = 0x04	None
2	Packet #	This number increments by 1 for each sample. Rolls over at 0xFF to 0x00.
6:3	Timestamp	1 LSB = 1.00 microseconds (Previously 1.33. Updated 2013.07.18)
8:7	Phi (Roll)	1 LSB = 0.1degree
10:9	Theta (Pitch)	1 LSB = 0.1degree
12:11	Rho (Compass)	1 LSB = 0.1degree
13	Packet END = 0x7E	None

Phi, Theta and Rho correspond to iPhi, iThe and iRho in the SV_COMMON structure definition.

7.1.5 Packet Type 5: Altitude and Temperature

This packet type is used to send altitude and temperature information to the Sensor Fusion Toolbox for Windows. The data is shown in the **Altimeter** tab (Fig 39), which is present only when the hardware/firmware support this feature.

Byte #	Function	Units
0	Packet Start = 0x7E	None
1	Packet Type = 0x05	None
2	Packet #	This number increments by 1 for each sample. Rolls over at 0xFF to 0x00.
6:3	Timestamp	1 LSB = 1.00 microseconds
10:7	Altitude	1 LSB = 0.001 meters
12:11	Temperature	1 LSB = 0.01 degree
13	Packet END = 0x7E	Use the same notations.

The Altitude field is derived from SV_1DOF_P_BASIC.fLPH. The temperature field is derived from SV_1DOF_P_BASIC.fLPT.

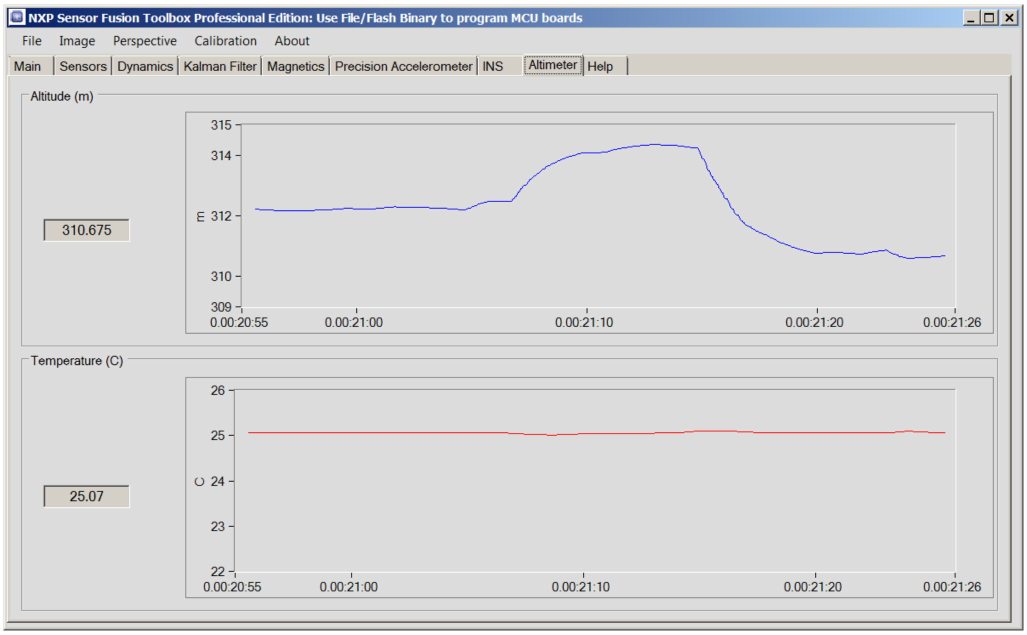


Fig 39. The Altimeter tab in the Sensor Fusion Toolbox for Windows

7.1.6 Reserved packet types

The following packet types are applicable to advanced algorithms that often change from release to release. Therefore, they are not documented for use outside of NXP.

7.1.6.1 Packet Type 6: Magnetic

Used to transmit samples from the magnetic constellation as well as computed calibration constants and fit parameters. Magnetic data is sent at a low bandwidth. Magnetic information displayed in the Sensor Fusion Toolbox may lag the actual status of the embedded code by some seconds.

7.1.6.2 Packet Type 7: Kalman Parameters

Used to transmit Kalman output and error terms.

7.1.6.3 Packet Type 8: Accelerometer Calibration

Used to read information displayed in the **Precision Accelerometer** tab of the Sensor Fusion Toolbox for Windows.

7.1.6.4 Packet Types 9, 10 & 11

Reserved for application-specific data

7.2 Toolbox to Freedom Development board

Up until version 2013.07.18 of the Fusion Toolbox, communication was strictly one way: from development platform to Android device. Versions released after that date include limited ability to configure the embedded board from the Sensor Fusion Toolbox. The command protocol is subject to change.

Possible 4-byte commands are shown below. Substitute a space for each "#" to enforce the 4-byte width.

ALT+	Altitude/Temperature packet on
ALT–	Altitude/Temperature packet off (default)[8]
DB+#	debug packet on (default) (transmitted via "Options Menu->Enable debug")
DB-#	debug packet off (transmitted via "Options Menu->Disable debug")
Q3##	transmit 3-axis accelerometer quaternion (tilt) in standard packet
Q3M#	transmit magnetometer quaternion (auto compass) in standard packet
Q3G#	transmit gyro-based quaternion in standard packet
Q6MA	transmit 6-axis mag/accel quaternion in standard packet (transmitted when "Remote mag/accel" is selected on the Source/Algorithms spinner)
Q6AG	transmit 6-axis accel/gyro quaternion in standard packet (transmitted when "Remote accel/gyro" is selected on the Source/Algorithms spinner)
Q9##	transmit 9-axis quaternion in standard packet (default) (transmitted when "Remote 9axis" is selected on the Source/Algorithms spinner)
RPC+	Roll/Pitch/Compass packet on
RPC–	Roll/Pitch/Compass packet off (default)
RST#	Sensor Fusion soft reset (resets all sensor fusion data structures)
VG+#	Virtual gyro packet on
VG-#	Virtual gyro packet off (default)
RINS	Reset INS inertial navigation velocity and position
SVAC	save all calibrations to Kinetis flash
SVMC	save magnetic calibration to Kinetis flash
SVYC	save gyroscope calibration to Kinetis flash
SVGC	save precision accelerometer calibration to Kinetis flash

[8]The debug packet must be enabled for firmware version number and sysTick counts to be properly displayed by the Toolbox.

ERAC	erase all calibrations from Kinetis flash
ERMC	erase magnetic calibration from Kinetis flash
ERYC	erase gyro offset calibration from Kinetis flash
ERGC	erase precision accelerometer calibration from Kinetis flash
180X	apply 180 degree perturbation in X
180Y	apply 180 degree perturbation in Y
180Z	apply 180 degree perturbation in A
M90X	apply –90 degree perturbation in X
P90X	apply +90 degree perturbation in X
M90Y	apply –90 degree perturbation in Y
P90Y	apply +90 degree perturbation in Y
M90Z	apply –90 degree perturbation in Z
P90Z	apply +90 degree perturbation in Z
PA*xx*	where *xx* = 00 through 11, return average precision accel location

Note that some of the commands above can request that the embedded board perform computations in a specific way. Confirm that the proper operation has taken place by checking the Flags field in Packet type 1.

8. Odds & Ends

8.1 ANSI C

The Sensor Fusion Library software is written according to the ANSI C90 standard and does not use any of the ANSI C99 or C11 extensions.

Integers are a mixture of standard C long (four byte) integers with typedef as int32_t and C short (2 byte) integers with typedef as int16_t. Floating point variables are all single precision of size four bytes.

8.2 Floating point libraries

The Sensor Fusion Library software uses single precision floating point arithmetic. C functions, like **sqrt()**, which return a double are immediately cast back into single precision.

Floating point arithmetic is performed using software emulation on the processors with integer cores or directly on the FPU by processors with an internal FPU.

8.3 Numerical accuracy

The Sensor Fusion Library software is numerically accurate to the limits of single precision floating point arithmetic.

Small angle approximations are only used when the relevant angle is, in fact, small so no accuracy is lost compared to using the standard libraries. For example, sines and cosines of small angles are computed with a MacLaurin series to give similar accuracy to the standard sine and cosine libraries but at lower computational expense.

Conditions that lead to inaccurate results are detected and the results computed using an alternative algorithm suitable for those cases where the primary algorithm fails. An example is the calculation of the orientation quaternion from a rotation matrix where the primary algorithm differencing elements across the diagonal approaches a 0/0 calculation. The alternative algorithm used near 180 degree rotations operates on the matrix diagonal elements instead.

The sensor fusion algorithms have been tested to be stable for tens of millions of 200 Hz iterations. Accumulation of rounding errors in the rotation quaternions or matrices is prevented by regular renormalization.

8.4 Error handling

NXP Application Note AN5017 provides a comprehensive overview of coordinate systems and orientation representations. This section simply introduces terms required by later sections of this user manual.

9. Debugging

9.1 Basic approach

The sensor fusion library has been downloaded thousands of times over the last few years. Developer questions often fall into common categories. Those categories have had a major impact in this rewrite of the user guide. This section discusses what to do when your application compiles and builds, but does not give expected results.

9.1.1 Start with a working solution

Even if you are porting the library to non-NXP hardware, you should ALWAYS have a working version based on NXP reference hardware on hand for comparison. And you should have a working copy of the Sensor Fusion Toolbox for Windows. Reasons for this recommendation include:

- Many developers new to sensor fusion do not fully understand the sensors involved. Access to a known good solution helps you through that learning curve and provides a valuable basis for comparison.
- Similarly, each fusion algorithm has its own set of tradeoffs to be considered. Experiment with a known good system before proceeding to make one of your own.
- The Sensor Fusion Toolbox is free, and Freedom development platforms are available for less than $100. There is no financial reason that any engineer starting in this area cannot have a working system for comparison.

9.1.2 Take a structured approach to debug

Always take a divide-and-conquer approach to debugging. Start by verifying your low-level drivers and work your way up the stack. Do not bother trying to debug the Kalman filter if you do not know that your sensor drivers and hardware abstraction layer (HAL) are correct. And do not debug your HAL until the sensor drivers are known good.

9.1.3 Community support

You should always first read this user guide. We have made every effort to anticipate common problems and include guidance here. But if that fails, NXP (and Freescale before it) has maintained an active user community for some years. The sensor fusion team at NXP checks it each workday and response times are usually measured in hours. NXP support will generally be limited to those using NXP hardware, but we also encourage developers to help one another regardless of the hardware platform involved. There is no cost to you to utilize the community.

When you submit a question or issue, please give us as much information as possible. Include:

- A brief description of your application.
- What tool chain(s) are you using?
- What MCU/sensors are you using?
- As much detail as possible about your question.

9.1.4 Are your performance expectation levels reasonable?

Table 25. Sensor strengths and weaknesses

Sensor	Strengths	Weaknesses
Accelerometer	• Inexpensive • Extremely low power • Very linear • Very low noise	• Measures the sum of gravity and acceleration. We need them separate.
Magnetometer	• The only sensor that can orient itself with regard to "North" • Insensitive to linear acceleration	• Subject to magnetic interference • Not "spatially constant"
Gyro	• Relatively independent of linear acceleration • Can be used to "gyro-compensate" the magnetometer	• Power hog • Long startup time • Zero rate offset drifts over time
Pressure Sensor	• The only stand-alone sensor that can give an indication of altitude	• Not well understood • A "relative" measurement • Subject to many interferences and environmental factors

Table 25 provides a high-level view of some of the tradeoffs to be considered in the sensor types used by this development kit. Developers new to sensors often do not fully understand the physics of the sensors and algorithms in question:

- There is no difference, from an accelerometer's perspective, between sustained acceleration and gravity. The two cannot be separated.
- Indoor magnetic fields vary tremendously over even short distances. These can easily mask the earth's magnetic field.
- We do not model all known sources of errors in our Kalman filters. We model those *most important to most applications*. There is always a tradeoff between accuracy, power, complexity and responsiveness. Kalman filters can be thought of as "least squares fit of a state-based model". The implication here is that any unmodeled sources of error will manifest as errors in those terms which ***are*** modeled.

Common comments/questions that arise:

Comment:	My application requires 1 degree accuracy relative to magnetic north. I can't seem to reach that with the library.
Response:	In a perfect environment, 10 degrees accuracy is relatively easy, 5 degrees is doable, 1 degree is a research project.
Question:	I am integrating linear acceleration to get velocity and then again to get position. It is not giving me the results I expect. Why?
Response:	Any error in the sensor output will cause the integration to quickly explode over time. See Section 2.8 for details.
Question:	My application is bolted to a stationary piece of equipment and does not give me expected response. Why?
Response:	Our Kalman filters are based upon standard equations *of motion*. If your board isn't moving, it is not collecting data to make the calculations.

9.2 Problems to expect with embedded debuggers

Embedded debuggers are a required tool in any developer's arsenal. However, they can easily lead you astray, especially if you attempt to single-step through your code.

- Compiler optimization can cause some variables to be completely optimized out and statements to be executed in slightly different order than you might expect. Turn off compiler optimization if you observe either problem.
- The fusion library programs sensors to sample at regular intervals. This continues even if you are at a breakpoint. Hardware FIFOs in the sensors will overflow. A good way to be sure you are getting correct values is to disable all breakpoints, set the program to run continuously, then dynamically set a breakpoint in your debugger while the program is running.
- Hardware faults can be especially hard to debug. Single stepping through code to find the root cause can be an exercise in futility. When using ARM processors, make sure you understand the various CPU registers, which can provide clues to the root cause. ARM®Keil® Application Note 209: *Using Cortex-M3 and Cortex-M4 Fault Exception* is useful in this regard. The old approach of simply commenting out portions of your code to isolate problems is still one of the best.

9.3 Are drivers called as scheduled?

In keeping with our bottoms-up methodology, start with your drivers. Are each of the initialization functions being called and is each returning a correct status? Initialization functions are called from within a simple loop inside function **initializeSensors()** in file sensor_fusion.c.

Likewise, are your sensor read functions operating correctly? They are called within function **readSensors()**, also in sensor_fusion.c.

If you are using an RTOS, is your read_task being invoked?

If your driver never returns, is your I^{2}C address correct? Are you using the correct I^{2}C port to access this particular sensor? Most MCUs have multiple I^{2}C ports. And it is possible to use more than one for even a single application.

If your driver results in a hard fault, make sure that the peripheral clocks are enabled for your I^{2}C module and associated port/GPIO.

9.4 Are serial communications working properly?

9.4.1 UART

By default, the UART port used to communicate with the Sensor Fusion Toolbox operates with the following settings:

- 115200 baud
- 8 bits data width
- No parity
- 1 stop bit

If you add to the communications protocol, be sure that you increase the size of the sUARTOutputBuffer variable. Failing to do so could result in a hardware fault as a result of buffer overflow.

9.4.2 I^{2}C/SPI

9.4.2.1 Do you have the correct I^{2}C address for each peripheral?

Use of an incorrect I^{2}C address can cause a driver to hang while waiting for a response from a nonexistent peripheral on the bus. Many NXP sensors can be pin programmed for multiple addresses. Check your schematic and sensor datasheet to ensure you have the right value.

9.4.2.2 Can you read the whoAmI register?

Well written sensor drivers should check the value of the sensor's whoAmI register during initialization. Check the status returned from the initialization function to ensure it found the expected sensor.

9.4.3 Are you using the correct Arduino pins for I2C?

When using the FRDM-STBC-AGM01 board, note J6 and J7 jumpers near the bottom of the board. These let you select between two different sources of I^{2}C signals. These must be set to the I^{2}C1 settings for FRDM boards and the I^{2}C0 settings for the LPCXpresso54114.

9.4.4 Issues specific to wired interfaces

Wired communications over USB make use of the virtual serial port capabilities of the OpenSDA interface used to program Kinetis devices. You must have the correct Windows driver for whichever version of OpenSDA bootloader you have installed on your board. The website nxp.com/opensda details options for all NXP Freedom Development Platforms.

Note: *One of the authors has a PC which consistently refuses to connect using the MBED CMSIS-DAP bootloader. Reprogramming the board bootloader to use the Segger version of OpenSDA fixes the problem.*

9.4.5 Issues specific to wireless interfaces

The FRDM-FXS-MULT2-B boards provide the option of wireless communication via standard Bluetooth. Make sure that the BT Power Jumper shown in Fig 2 is in place.

9.4.5.1 Bluetooth Pairing

Regardless of whether you are using the Windows or Android version of the Sensor Fusion Toolbox, you will need to pair your board to your device before doing anything else. Each board has a unique Bluetooth ID of the form BlueRadios*xxxxxx*, where *xxxxxx* are the last 6 hex digits on the 2nd line of the Bluetooth module label (“B“ in Fig 21).

9.4.5.2 Android Sensor Fusion Toolbox topics

Make sure you enable Bluetooth communications in Android application by following the procedure outlined in steps 10 through 12 of Section 3.5. You will not be able to communicate with your board until you complete these steps.

9.4.5.3 GUI latency

Conceptually, the sensor fusion library computes 1 orientation per iteration of each filter. This can be represented as a quaternion in fixed point format using only 8 bytes. The UART serial port interface used by the GUI runs at 115,200 baud. You might think that bandwidth would not be an issue. But the library transmits a lot of information to assist in the debug effort. The GUI update rate is throttled to 40 Hz or less to accommodate these needs. So even if you run fusion at 200 Hz, you will still see GUI updates at only 40X per second.

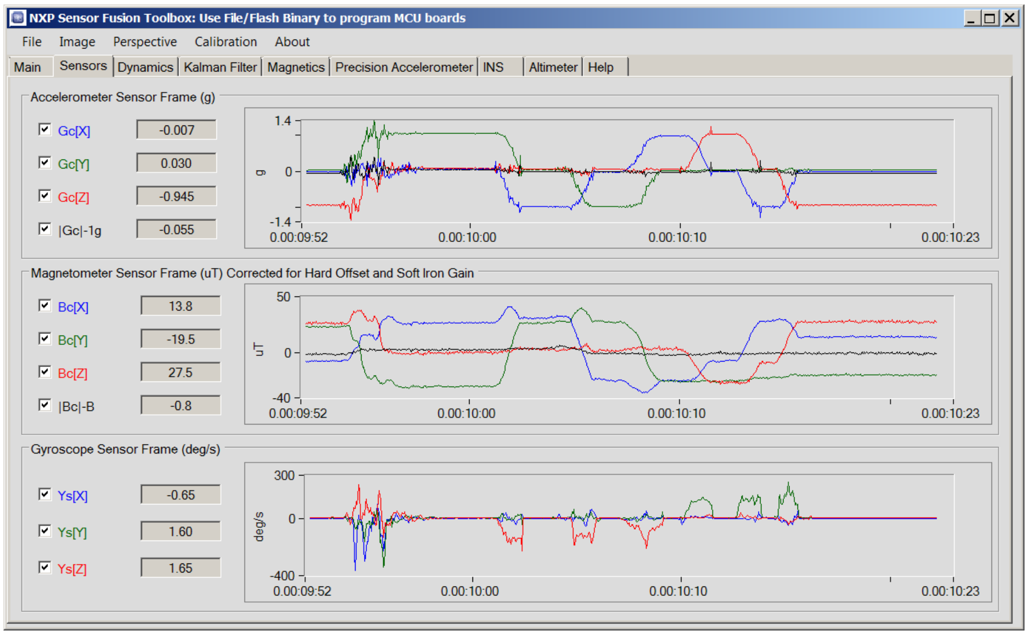
Some diagnostics, notably magnetic calibration, require a lot of data to be transmitted to the GUI. These are transmitted at a lower data rate, which can cause apparent delays in the display. For instance, if you wave a magnet over the magnetic sensor, it may take several seconds to see the magnetic constellations update accordingly. This is normal. The delay is only in the transmission. Algorithms running on the MCU itself do respond in a timely manner.

9.5 Do drivers return reasonable values?

This is where having the Sensor Fusion Toolbox for Windows and a working reference design (see section 9.1.1) really come in handy.

9.5.1 Accelerometers

Checking accelerometer readings is easy. Connect your system to the Sensor Fusion Toolbox and click on the “Sensors” tab. For any of the standard 6 orientations, two axes should have approximately zero values and the other will be ±1*g*. Fig 40 shows the case where we have rotated a board through all six orientations.


NXP Sensor Fusion Toolbox: Use File/Flash Binary to program MCU boards
File Image Perspective Calibration About
Main Sensors Dynamics Kalman Filter Magnetics Precision Accelerometer INS Altimeter Help
g
uT
deg/s
0.00:09:52
0.00:10:00
0.00:10:10
0.00:10:23

Accelerometer Sensor Frame (g)

Gc[X]	-0.007
Gc[Y]	0.030
Gc[Z]	-0.945
|Gc|-1g	-0.055

Magnetometer Sensor Frame (uT) Corrected for Hard Offset and Soft Iron Gain

Bc[X]	13.8
Bc[Y]	-19.5
Bc[Z]	27.5
|Bc|-B	-0.8

Gyroscope Sensor Frame (deg/s)

Ys[X]	-0.65
Ys[Y]	1.60
Ys[Z]	1.65

Fig 40. The Sensors tab in the Sensor Fusion Toolbox for Windows

Fig 41 provides a pictorial reference for an Android build on a K64F/Mult2-B board stackup. Remember that these values change a function of the frame of reference. See Table 13.

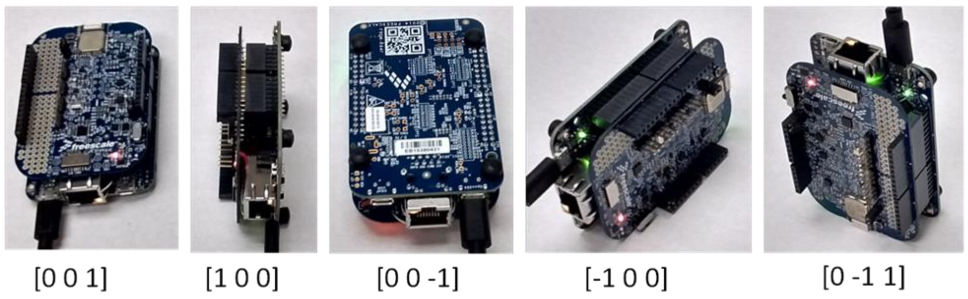


Fig 41. Accelerometer readings for 5 of the 6 Standard Orientations for an Android build

9.5.2 Magnetometers

Recall the discussions in Section 5.4.3, and the **Magnetics** tab of the Sensor Fusion Toolbox as shown in Fig 36. As you rotate your board, the **Calibrated Display** should show circular X-Y, Y-Z and Z-X projections. ±30 to ±60 μT is the expected range for the sample distributions.

If the calibrated displays are elliptical, this indicates an extreme case of soft-iron magnetic interference. This results from the presence of some highly ferrous material very close to the sensor. The NXP magnetic calibration routines can usually correct for all but extreme cases of magnetic interference fixed spatially with respect to the sensor.

It is common to see errors in the absolute values of the geomagnetic field intensity and inclination angle. These are “secondary variables” falling out of the magnetic calibration routines and, at least indoors, are rough estimates at best.

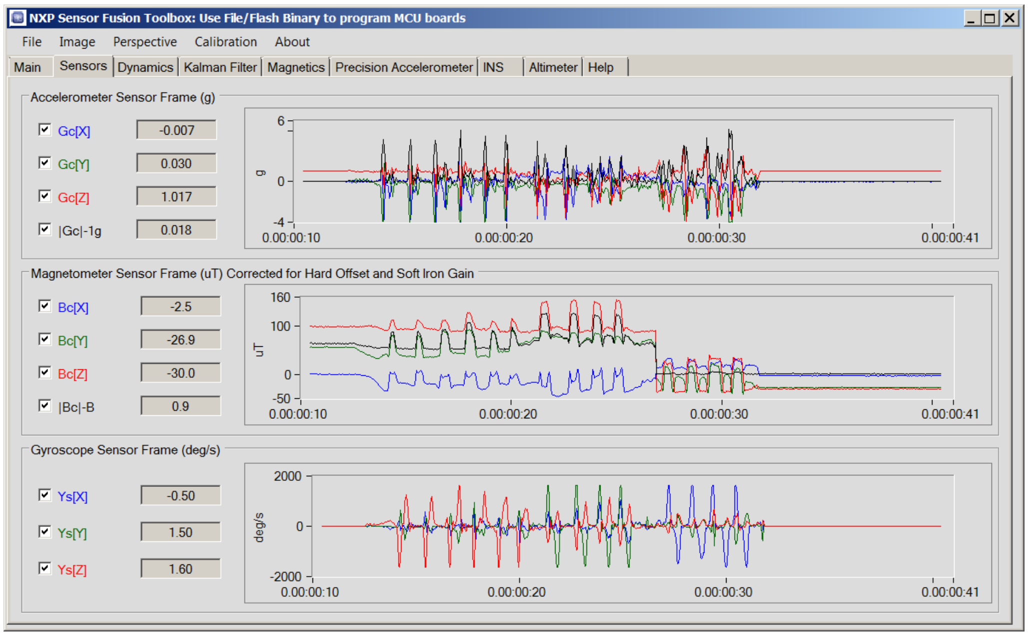
9.5.3 Gyroscopes

Gyros are a bit harder to evaluate, as they are rate sensors. If your board is stationary, the gyro should read near zero on all channels. There are two tests you can quickly run. For both, you want to select the “Rotation” algorithm selection on the Sensor Fusion Toolbox main tab.

9.5.3.1 Dynamic range

Here you want to hold the board in your hand and make quick twists of the wrist to induce a signal in each of the three axes. Looking for transitions in the expected directions, as shown in the Gyroscope Sensor Frame below. Try to turn the board as fast as you can to see if you can reach the maximum rate supported by the gyroscope.

This method can be somewhat hazardous to your wrist. If you have a record player, a better way is to simply put the board on top and let it spin continuously. Lazy Susans and bicycle wheels work also.



Accelerometer Sensor Frame (g)

Gc[X]	-0.007
Gc[Y]	0.030
Gc[Z]	1.017
|Gc|-1g	0.018

Magnetometer Sensor Frame (uT) Corrected for Hard Offset and Soft Iron Gain

Bc[X]	-2.5
Bc[Y]	-26.9
Bc[Z]	-30.0
|Bc|-B	0.9

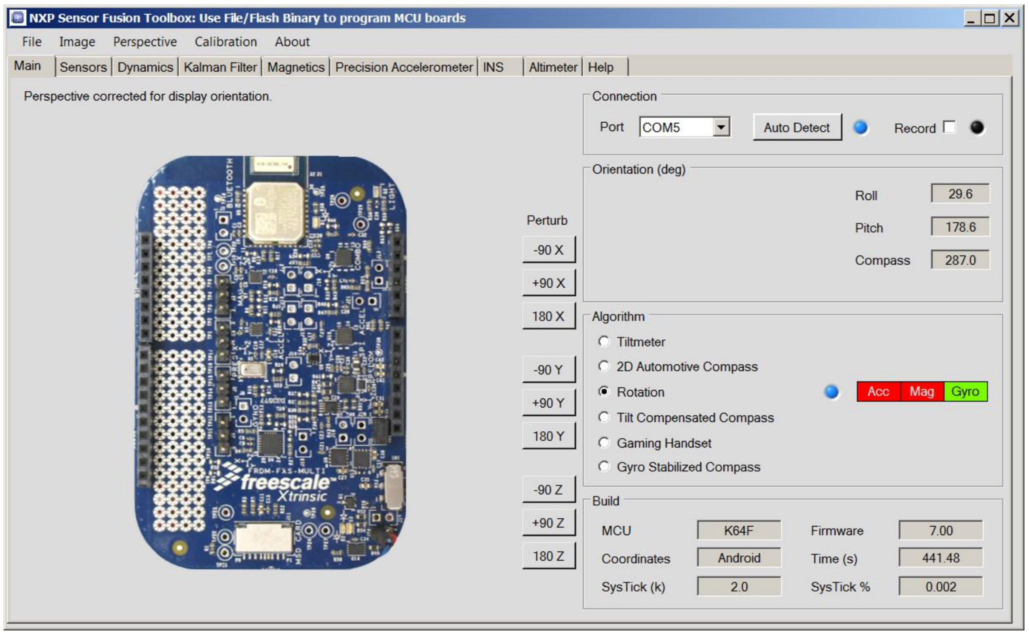
Gyroscope Sensor Frame (deg/s)

Ys[X]	-0.50
Ys[Y]	1.50
Ys[Z]	1.60

Fig 42. Manual Gyroscope Test

9.5.3.2 Integrated orientation

Changes in displayed board rotation should track changes in the actual board location when using the “Rotation” algorithm selection. To make it easier, set your board on a table top and reset the display using the Perspective->Correct for Display Orientation function to realign the display as shown in Fig 43.


NXP Sensor Fusion Toolbox: Use File/Flash Binary to program MCU boards
File Image Perspective Calibration About
Main Sensors Dynamics Kalman Filter Magnetics Precision Accelerometer INS Altimeter Help
Auto Detect
Perturb -90 X +90 X 180 X -90 Y +90 Y 180 Y -90 Z +90 Z 180 Z
Acc Mag Gyro

Perspective corrected for display orientation.

Connection: Port COM5, Record ☐

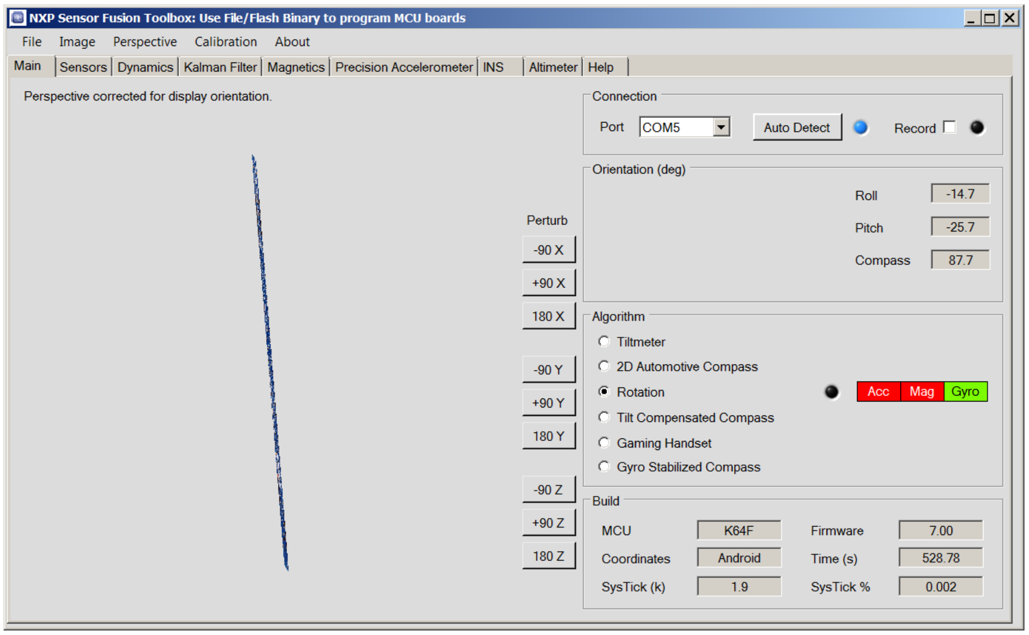
Orientation (deg): Roll 29.6, Pitch 178.6, Compass 287.0

Algorithm: ○ Tiltmeter, ○ 2D Automotive Compass, [✓ Rotation], ○ Tilt Compensated Compass, ○ Gaming Handset, ○ Gyro Stabilized Compass

Build			
MCU	K64F	Firmware	7.00
Coordinates	Android	Time (s)	441.48
SysTick (k)	2.0	SysTick %	0.002

Fig 43. Starting gyro orientation test

Now quickly rotate the board 90 degrees. The display should now look something like Fig 44. The implication is that the board has rotated the expected 90 degrees. Errors in gyro range selection or fusion time interval manifest as errors in rotation here.


NXP Sensor Fusion Toolbox: Use File/Flash Binary to program MCU boards
File Image Perspective Calibration About
Main Sensors Dynamics Kalman Filter Magnetics Precision Accelerometer INS Altimeter Help
Auto Detect
Perturb -90 X +90 X 180 X -90 Y +90 Y 180 Y -90 Z +90 Z 180 Z
Acc Mag Gyro

Perspective corrected for display orientation.

Connection: Port COM5, Record ☐

Orientation (deg): Roll -14.7, Pitch -25.7, Compass 87.7

Algorithm: ○ Tiltmeter, ○ 2D Automotive Compass, [✓ Rotation], ○ Tilt Compensated Compass, ○ Gaming Handset, ○ Gyro Stabilized Compass

Build			
MCU	K64F	Firmware	7.00
Coordinates	Android	Time (s)	528.78
SysTick (k)	1.9	SysTick %	0.002

Fig 44. Gyro orientation test after 90 degrees board rotation

For instance, if the board displays at 45 degrees instead of 90, it indicates that either the gyro is set at twice the expected range, or the fusion time interval is half the expected value.

9.5.4 Version compatibility issues?

Each version of the Sensor Fusion Library is typically released concurrent with an updated version of the Sensor Fusion Toolbox for Windows. The version 5.00 library and matching Toolbox are available at nxp.com/sensorfusion. Version 7.xx of the sensor fusion library is released as part of the MCUXPRESSO SDK builds created by the MCUXpresso Config Tool. The matching Toolbox will be available, and clearly labeled as for use with V7.xx, at nxp.com/sensorfusion. The format of the magnetic packet changed with the V6.00 library. So some features on the **Magnetics** tab will not work if you attempt to mix and match.

The Android version of the Sensor Fusion Toolbox does not support line plots, Kalman analysis or magnetic calibration features. It is backward compatible with Version 5.00 of the sensor fusion library.

9.6 Have you tried different compiler optimization levels?

MCUs with limited clock rates and/or limited memory may have trouble running the some algorithms. Make sure you try the high optimization settings of your compiler before concluding you actually have a problem.

9.7 Symptoms associated with RTOS stack problems

Most problems encountered with FreeRTOS relate to the configuration of stack and heap sizes in FreeRTOSConfig.h. If properly configured, the call to vTaskStartScheduler() in main() should never return.

9.8 Magnetic Interference

We have discussed magnetic interference numerous times in this text, but it bears repeating: You should assume that indoor magnetic environments vary significantly over even short distances. If your development board is near a laptop computer or telephone, it will encounter distorted fields. This is not a sensor problem, it is an environmental problem. Sensor fusion can help to alleviate, but not cure, this issue.

10. Theory: orientation representations

Sections 10.1 and 10.2 are variations of two postings (Orientation Representations, Part 1 & Part 2) which initially appeared on the Freescale (now NXP) Embedded Beat blog site in 2012. To have a “happy experience” integrating sensor fusion into an application, you need to understand the underlying theory presented here.

10.1 Part 1: Euler angles and rotation matrices

10.1.1 Rotation matrices

If you look at the software source code at the heart of these AR and VR apps, you will discover a common component: a mathematical model relating the position and orientation of your portable device within an earth frame of reference. Perhaps the three most common are Euler angles, rotation matrices and quaternions. This section focuses on the Euler angles and rotation matrices. Quaternions will be covered in the next section.

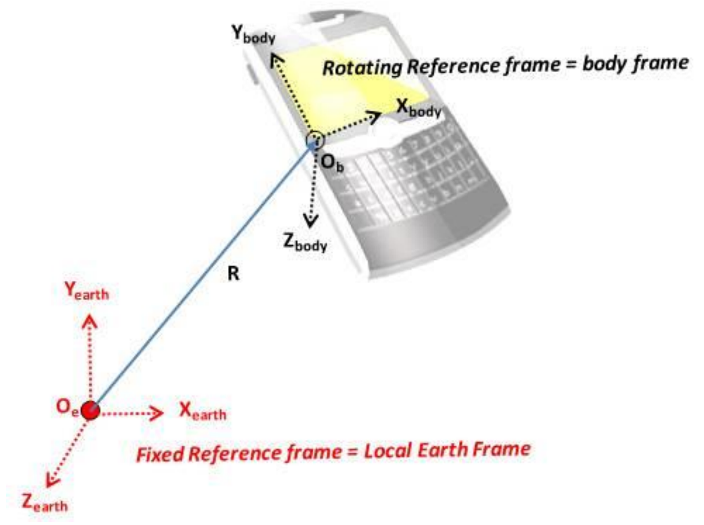

Y_{body}
Rotating Reference frame = body frame
X_{body}
O_b
Z_{body}
R
Y_{earth}
O_e
X_{earth}
Fixed Reference frame = Local Earth Frame
Z_{earth}

Fig 45. Local earth versus the body frame

Let us arbitrarily choose the phone screen X,Y coordinate system as our rotating frame of reference for this discussion. You can intuitively see that any X,Y point (Z assumed zero) on the phone screen will map to some X, Y, Z point in the earth frame of reference. It is also clear that the Cartesian axes in the phone (or body) frame of reference will only rarely align themselves with the local earth frame axes. Since the position offset, R, does not affect orientation, we can collapse the figure down to that shown in the figure below.

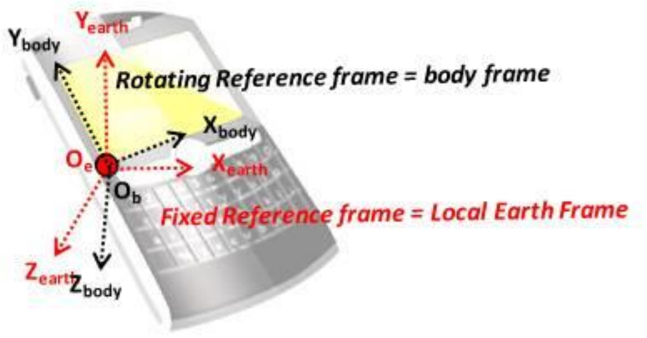

Y_{earth}
Y_{body}
Rotating Reference frame = body frame
X_{body}
O_e
X_{earth}
O_b
Fixed Reference frame = Local Earth Frame
Z_{earth}
Z_{body}

Fig 46. Using a common origin for both earth and body frames

Now let us remove the phone from the picture and focus just on points in the X-Y plane. The figure below shows the earth frame rotated into the body frame by an angle Psi (ψ).

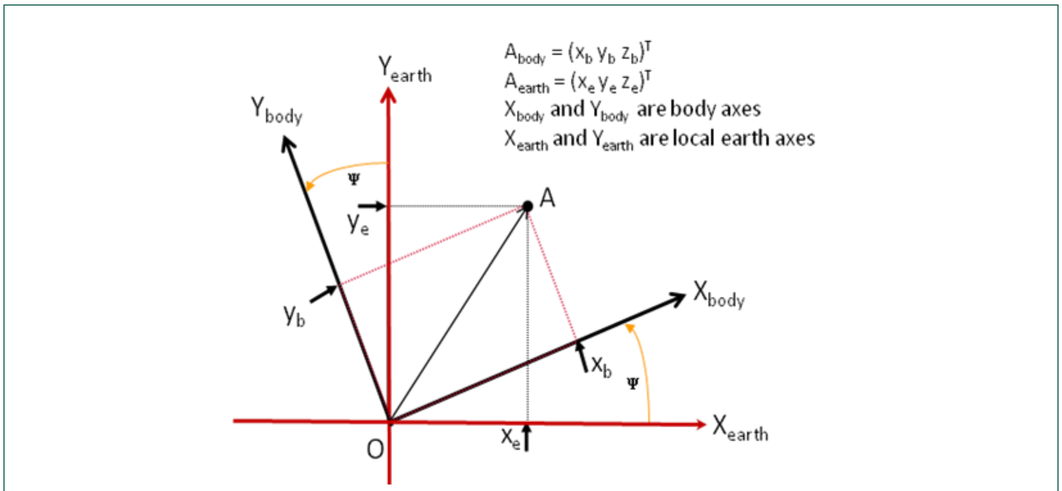

$A_{body} = (x_b\ y_b\ z_b)^T$
$A_{earth} = (x_e\ y_e\ z_e)^T$
X_{body} and Y_{body} are body axes
X_{earth} and Y_{earth} are local earth axes
Y_{earth}
Y_{body}
Ψ
A
y_e
y_b
X_{body}
x_b
Ψ
X_{earth}
O
x_e

Fig 47. The X-Y plane illustrates rotation from earth into body frame about the Z-axis

You probably first saw a figure like this in your high school geometry or trig course. It allows us to map any point “A” in any standard X,Y Cartesian system to any other X,Y Cartesian system, which is rotated from it by some angle Psi (ψ), with a simple linear transformation. To see how, let us deconstruct that figure using a number of right triangles.

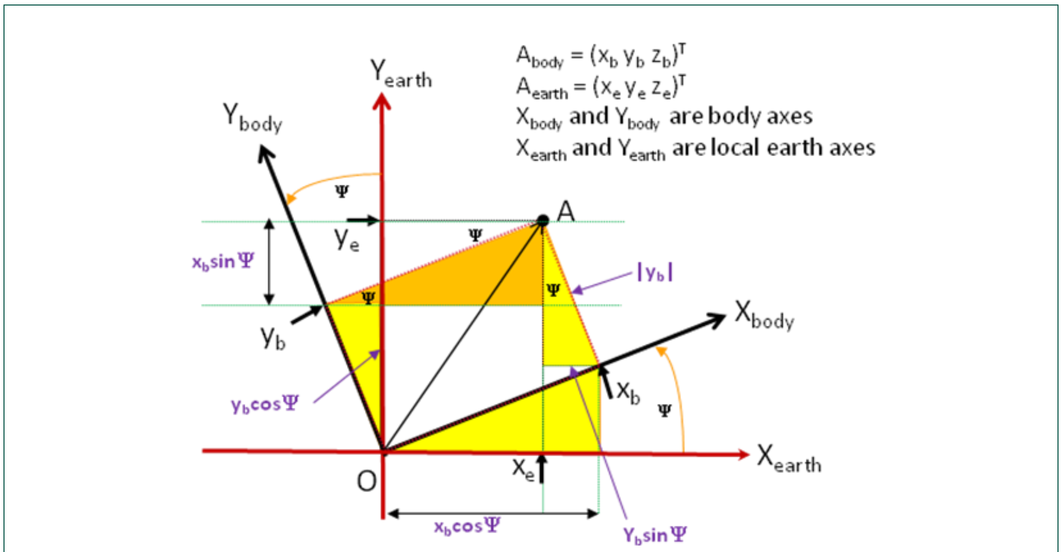

$A_{body} = (x_b\ y_b\ z_b)^T$
$A_{earth} = (x_e\ y_e\ z_e)^T$
X_{body} and Y_{body} are body axes
X_{earth} and Y_{earth} are local earth axes
Y_{earth}
Y_{body}
Ψ
A
y_e
$x_b \sin\Psi$
$|y_b|$
y_b
X_{body}
$y_b \cos\Psi$
x_b
X_{earth}
O
x_e
$x_b \cos\Psi$
$Y_b \sin\Psi$

Fig 48. Physical justification for terms in the rotation matrix

If you study the figure above, you can see that the rotation angle, ψ, is present in each of the four right triangles we added. Additionally, the hypotenuse of each triangle is either x_b or y_b. Given that information, we can see how to compute x_e and y_e from x_b and y_b:

$x_e = x_b \cos(Y) - y_b \sin(Y)$ (Eqn. 1)

$y_e = x_b \sin(Y) + y_b \cos(Y)$ (Eqn. 2)

In matrix form:

$\mathbf{A}_{earth} = \mathbf{C}(-\psi)\ \mathbf{A}_{body}$ (Eqn. 3)

The inverse of which is:

$\mathbf{A}_{body} = \mathbf{C}(\psi)\ \mathbf{A}_{earth}$ (Eqn. 4) where $\mathbf{A}_{earth}$ and $\mathbf{A}_{body}$ are of the form $[x\ y]^T$ and:

$$\mathbf{C}(\psi) = \mathbf{C}_{\psi} = \begin{bmatrix} cos\psi & sin\psi \\ -sin\psi & cos\psi \end{bmatrix} \quad \text{(Eqn. 5)}$$

$$\mathbf{C}(-\psi) = \mathbf{C}_{-\psi} = \begin{bmatrix} cos\psi & -sin\psi \\ sin\psi & cos\psi \end{bmatrix} \quad \text{(Eqn. 6)}$$

All other relationships included in this discussion can be similarly mapped to a diagram of the rotation. The analysis also extends naturally to three dimensions. If $\mathbf{A}_{earth}$ and $\mathbf{A}_{body}$ are of the form $[x\ y\ z]^T$, then:

$$\mathbf{C}(\psi) = \mathbf{C}_{\psi} = \begin{bmatrix} cos\psi & sin\psi & 0 \\ -sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{(Eqn. 7)}$$

$$\mathbf{C}(-\psi) = \mathbf{C}_{-\psi} = \begin{bmatrix} cos\psi & -sin\psi & 0 \\ sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{(Eqn. 8)}$$

We will use the three-dimensional form in the remainder of this discussion. In both two- and three-dimensional cases, **C**(ψ) and **C**(-ψ) are known as *rotation matrices.*

Notice that **C**(ψ) is the transpose of **C**(-ψ), and vice-versa: $\mathbf{C}(\psi)^T = \mathbf{C}(-\psi)$. This is a special property of all rotation matrices. You can reverse the sense of rotation simply by taking the transpose of the original matrix.

$$\mathbf{C}(\psi)\ \mathbf{C}(-\psi) = \mathbf{I}_{3x3}$$

Where $\mathbf{I}_{3x3}$ is simply the identity matrix:

$$\mathbf{I}_{3x3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This also implies that the inverse of a rotation matrix is simply its own transpose:

$$\mathbf{C}(\psi)^{-1} = \mathbf{C}(\psi)^T$$

Similar relationships hold for rotations in the X-Z (about the Y-axis) plane:

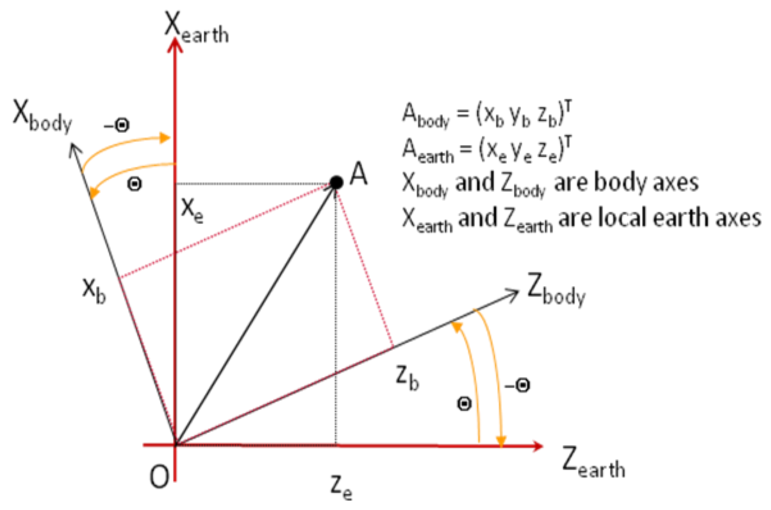

X_{earth}
X_{body}
$-\Theta$
Θ
A
x_e
x_b
$A_{body} = (x_b\, y_b\, z_b)^T$
$A_{earth} = (x_e\, y_e\, z_e)^T$
X_{body} and Z_{body} are body axes
X_{earth} and Z_{earth} are local earth axes
Z_{body}
z_b
$-\Theta$
Θ
O
z_e
Z_{earth}

Fig 49. The X-Z plane illustrates rotation about the Y-axis

$x_e = x_b \cos(\theta) + z_b \sin(\theta)$ (Eqn. 12)

$z_e = - x_b \sin(\theta) + z_b \cos(\theta)$ (Eqn. 13)

$\mathbf{A}_{\mathbf{earth}} = \mathbf{C}(-\theta)\, \mathbf{A}_{\mathbf{body}}$ (Eqn. 14)

$\mathbf{A}_{\mathbf{body}} = \mathbf{C}(\theta)\, \mathbf{A}_{\mathbf{earth}}$ (Eqn. 15)

$$\mathbf{C}(-\Theta) = \mathbf{C}_{-\theta} = \begin{bmatrix} cos\Theta & 0 & sin\Theta \\ 0 & 1 & 0 \\ -sin\Theta & 0 & cos\Theta \end{bmatrix}$$ (Eqn. 16)

$$\mathbf{C}_{\theta} = \begin{bmatrix} cos\Theta & 0 & -sin\Theta \\ 0 & 1 & 0 \\ sin\Theta & 0 & cos\Theta \end{bmatrix}$$ (Eqn. 17)

and for rotations in the Z-Y (about the X-axis) plane:

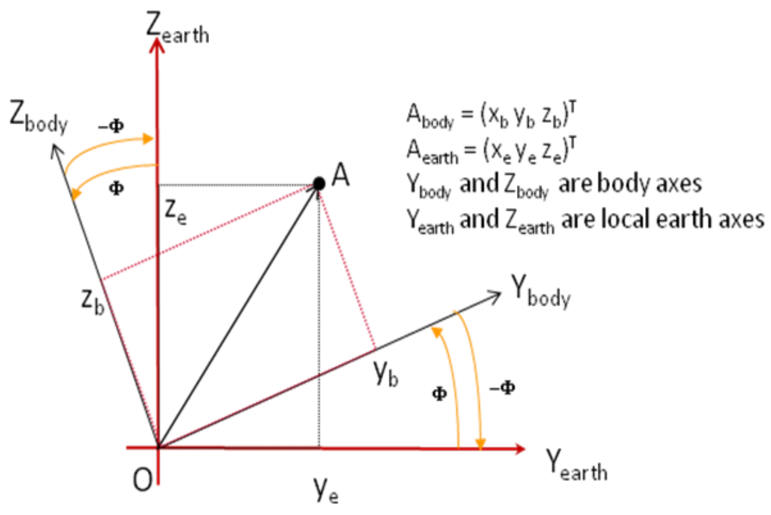

Z_{earth}
Z_{body}
$-\Phi$
Φ
A
z_e
z_b
$A_{body} = (x_b\, y_b\, z_b)^T$
$A_{earth} = (x_e\, y_e\, z_e)^T$
Y_{body} and Z_{body} are body axes
Y_{earth} and Z_{earth} are local earth axes
Y_{body}
y_b
Φ
$-\Phi$
O
y_e
Y_{earth}

Fig 50. The Y-Z plane illustrates rotation about the X-axis

$y_e = y_b \cos(\Phi) - z_b \sin(\Phi)$ (Eqn. 18)

$z_e = y_b \sin(\Phi) + z_b \cos(\Phi)$ (Eqn. 19)

$\mathbf{A}_{\mathbf{earth}} = \mathbf{C}(-\Phi)\ \mathbf{A}_{\mathbf{body}}$ (Eqn. 20)

$\mathbf{A}_{\mathbf{body}} = \mathbf{C}(\Phi)\ \mathbf{A}_{\mathbf{earth}}$ (Eqn. 21)

$$\mathbf{C}(-\Phi) = \mathbf{C}_{-\Phi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\Phi & -sin\Phi \\ 0 & sin\Phi & cos\Phi \end{bmatrix}$$ (Eqn. 22)

$$\mathbf{C}_{\Phi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\Phi & sin\Phi \\ 0 & -sin\Phi & cos\Phi \end{bmatrix}$$ (Eqn. 23)

Phi, Theta and Psi (Φ, θ and ψ) rotations <u>in sequence</u> can map a point in any right-hand-rule (RHR) three-dimensional space into any other RHR three-dimensional space.

A rotation from earth by body frame about Z, then Y then X axes (the “aerospace” sequence) is represented by:

$$\mathbf{C}_{\mathbf{RPY}} = \mathbf{C}_{\Phi}\mathbf{C}_{\Theta}\mathbf{C}_{\Psi} = \begin{bmatrix} cos\Psi cos\Theta & cos\Theta sin\Psi & -sin\Theta \\ sin\phi sin\Theta cos\Psi - cos\phi sin\Psi & sin\phi sin\Theta sin\Psi + cos\phi cos\Psi & sin\phi cos\Theta \\ cos\phi sin\Theta cos\Psi + sin\phi sin\Psi & cos\phi sin\Theta sin\Psi - sin\phi cos\Psi & cos\phi cos\Theta \end{bmatrix}$$

(Eqn 24)

The composite rotation matrix is computed simply by multiplying the three individual matrices in the specified order. Consistent with the discussion above, the inverse of this expression is:

$$\mathbf{C}_{\mathbf{YPR}} = \mathbf{C}_{-\Psi}\mathbf{C}_{-\Theta}\mathbf{C}_{-\Phi} = \begin{bmatrix} cos\Psi cos\Theta & sin\phi sin\Theta cos\Psi - cos\phi sin\Psi & cos\phi sin\Theta cos\Psi + sin\phi sin\Psi \\ cos\Theta sin\Psi & sin\phi sin\Theta sin\Psi + cos\phi cos\Psi & cos\phi sin\Theta sin\Psi - sin\phi cos\Psi \\ -sin\Theta & sin\phi cos\Theta & cos\phi cos\Theta \end{bmatrix}$$

(Eqn. 25)

10.1.2 Euler Angles

Collectively, Φ, θ and ψ are known as *Euler angles*. You may also see Φ, θ and ψ referred to as *roll*, *pitch* and *yaw* respectively. The subscript “RPN” in the expression above refers to roll-pitch-yaw and “YPR” refers to yaw-pitch-roll.

Euler angles are sometimes subdivided into “Tait-Bryan” angles (in which rotations occur about all three axes) and “proper” Euler angles (in which the first and third axes of rotation are the same). Regardless of which type you use, it is important to specify the order of the rotations, which IS significant. In the table below, the right-most rotation is performed first, consistent with the matrix operations that will be required to implement the rotation. Possible variants are:

Alpha	**Angles**	**Comments**	
YRP	ψ-Φ-θ	Tait-Bryan	All of these

YPR	ψ-θ-Φ	angles (AKA Nautical or Cardan angles)	are sometimes referred to simply as “Euler Angles”
PYR	θ-ψ-Φ		
PRY	θ-Φ-ψ		
RYP	Φ-ψ-θ		
RPY	Φ-θ-ψ		
RYR	Φ-ψ-Φ	Proper Euler Angles	
RPR	Φ-θ-Φ		
PYP	θ-ψ-θ		
PRP	θ-Φ-θ		
YRY	ψ-Φ-ψ		
YPY	ψ-θ-ψ		

A full discussion of Euler angles is beyond the scope of this article. But some key points you need to take away are:

- Again, order of rotation matters!
- There are almost as many notations for Euler angles as the references in which they appear. Be specific in *you*r notation.
- There may be multiple Euler angle combinations which map to the same physical rotation. They are not unique.
- There are typically limits on the range of Euler rotation for each axis (±π or ±π/2). These affect how the total rotation is spread across the three angles. They also tend to introduce discontinuities when Euler angles are computed over time.
- Each Euler triad will map to a 3x3 rotation matrix, which IS unique, in a manner similar to that shown in Equations 24 and 25 above.

If you define a “reference orientation” for any object, then you can define its current orientation as some rotation relative to that reference. Tracking orientation over time is then equivalent to tracking rotation from that reference over time. Because Euler angles are relatively easy to visualize, they enjoyed early popularity in a number of fields. But because of the shortcomings listed above, anyone who has used them extensively invariably learns to despise them.

Rotation matrices do not require a master’s degree to be able to use them. Anyone with a college freshman geometry class under their belt can figure them out. They do not have the ambiguities associated with Euler angles, and can be found at the heart of many algorithms. But you need nine numbers for each rotation in this form. That can chew up a lot of storage. A better representation is the “quaternion”, which we will explore in the next section.

10.1.3 References:

[1] Quaternions and Rotation Sequences, Jack B. Kuipers, Princeton University Press, 1999

[2] Euler Angles from the Wolfram Demonstrations Project by Frederick W. Strauch

[3] Diversified Redundancy in the Measurement of Euler Angles Using Accelerometers and Magnetometers, Chirag Jagadish and Bor-Chin Chang, Proceedings of the 46th IEEE Conference on Decision and Control, Dec. 2007

[4] "Euler Angles" at Wikipedia

10.2 Part 2: Quaternions

10.2.1 Discussion

In the previous section, we explored the use of rotation matrices and Euler angles. At the end of that discussion, we alluded to the fact that there might be more efficient ways of describing rotations. Let us start with the rotation of a simple rigid body, in this case a cylinder, as shown in the figure below. Here, the cylinder is rotated such that a point on its surface originally at "A" is rotated to point "B" in space

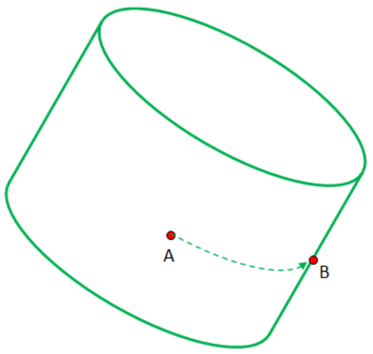


Fig 51. Rotation of a rigid body such that a reference point moves from "A" to "B"

For this simple case, we have kept the axis of rotation along the vertical axis of the cylinder as shown in the figure below. But that is not a requirement for the underlying mathematics to work. So long as we have a rigid body, we can always describe the rotation in the manner that follows.

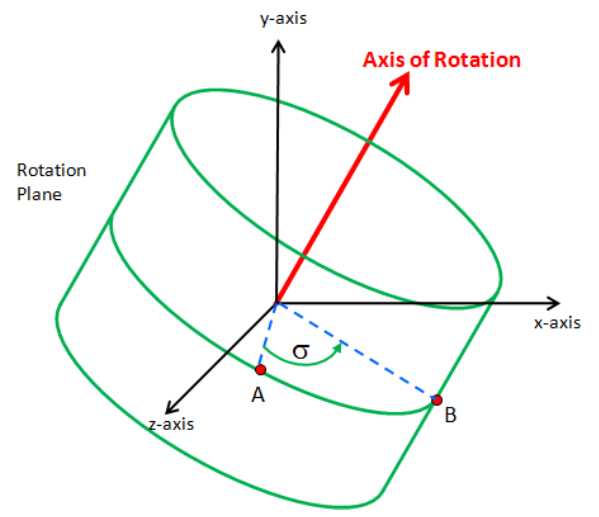


Fig 52. Overlay of Cartesian Coordinates onto System of Figure 1

Fig 53 deals with the same rotation, but focuses on the fact that we have a rotation plane that is perpendicular with the axis of rotation. The movement of the cylinder is a rotation equal of angle α, about the axis of rotation, where the point of interest is constrained to lie within the rotation plane.

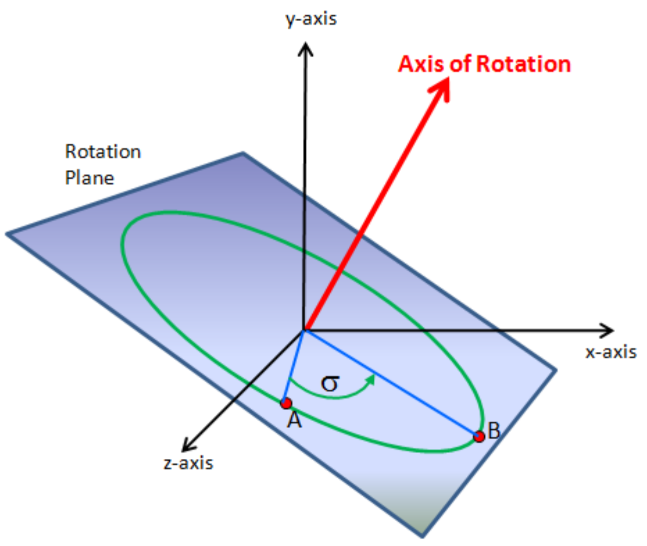


Fig 53. Looking at just the rotation plane and axis of rotation

The rotation is fully described by the three components of the normalized rotation axis and the rotation angle α, which may be in radians or degrees, depending upon the system in use. As an example, consider OpenGL ES graphics programming. This system is very popular on portable devices. We used it to program the **Device** screen in the Sensor Fusion Toolbox for Android. In OpenGL ES, you build up 3 dimensional objects as a collection of triangles, which can then be offset and/or rotated to change perspective. As an example, our cylinder might be crudely drawn as shown in Figure 4.

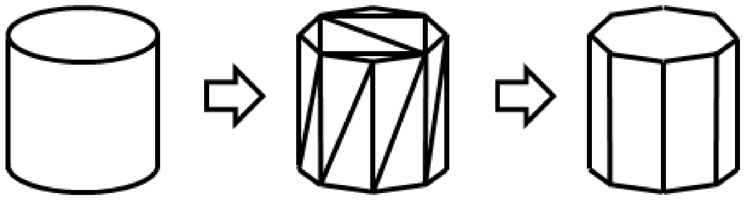


Fig 54. OpenGL ES Drawing of a Cylinder

In this case, we have modeled the top and bottom of the cylinder with 6 triangles each, and the other side is modeled using a total of 16 triangles arranged in a strip. OpenGL ES is optimized to draw such structures efficiently, and it is possible to then “render” textures onto the drawn surfaces. What is really neat is that once drawn, we get a reasonable approximation of the cylinder on the left of Fig 54 simply by doing a –30 degrees rotation about the Z-axis (presumed to be out of the page) using a <u>single</u> OpenGL ES instruction:

gl.glRotatef(–30.0f, 0.0f, 0.0f, 1.0f);

At this point, you’re probably thinking: “Yeah, that makes sense, but how does it work at the math level?” This is the where we need to introduce the concept of a quaternion. Conceptually, a quaternion encodes the same axis and angle as above. But for mathematical reasons it deals with half of the rotation angle as shown below.

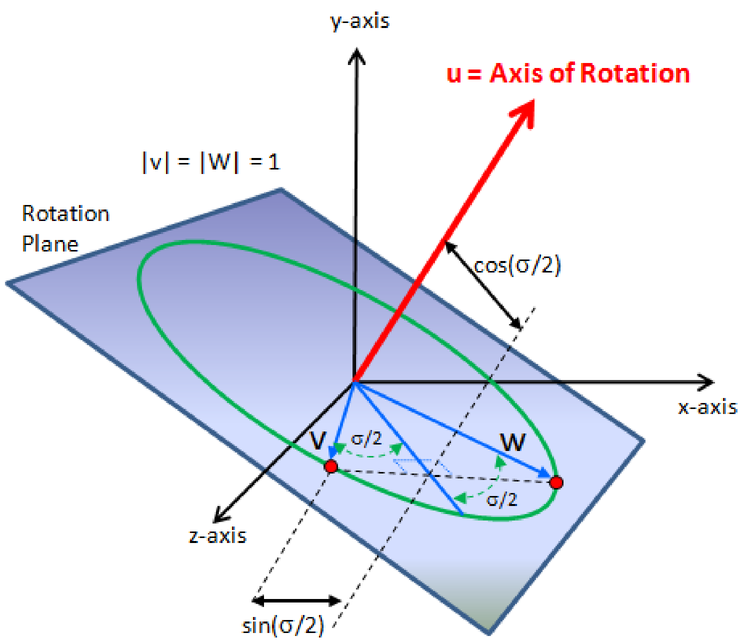


Fig 55. System of Figure 4 in terms of Quaternion components

Before overwhelming you with the underlying math, you should know that unless you are planning to implement your own quaternion utility library, you only need to know a few key points:

1. It takes four numbers to fully describe a quaternion (commonly q0 through q3).
2. Not all quaternions are rotation quaternions. Rotation quaternions have unit length ($q0^2 + q1^2 + q2^2 + q3^2 = 1$). The discussion below will be restricted to rotation quaternions.
3. These same rotations can be described using Euler angles, rotation matrices, etc. as discussed in the previous section. It is possible (and common) to translate between formats and use multiple formats. Rotation matrices have the advantage of always being unique. Euler angles are subject to gimbal lock and should not be used for internal calculations (only input/output of results).
4. You can rotate a vector **V** using a quaternion q using the equation: **W** = q**V**q* (quaternion products and complex conjugates are defined later)
5. A sequence of rotations represented by quaternions q1 followed by q2 can be collapsed into a single rotation simply by computing the quaternion product q=q2*q1 and then applying the rotation operator as above.

We will be presenting the mathematical definition first and without proof. If you really, REALLY want to know the underlying theory, let us suggest that you pick up a copy of Jack Kuiper's excellent text: *Quaternions and Rotation Sequences*. This appears to be, by far, the most extensive treatment on the topic, even while remaining very readable.

Notice that rotation quaternions deal with α/2, not α. We can define a rotation quaternion "q" in one of several equivalent fashions.

$q = (q0, q1, q2, q3)$ (Eqn.1)

$q = q0 + \mathbf{q}$, where $\mathbf{q} = \mathbf{i}q1 + \mathbf{j}q2 + \mathbf{k}q3$ (Eqn. 2)

$q = \cos(\alpha/2) + u \sin(\alpha/2)$, where u is the vector axis of rotation (Eqn. 3)

We use the quaternion form where q0 = cos(α/2). Some texts will reorder the quaternion components so that the vector portion **q** is contained in q0–2 and q3 = cos(α/2). Be sure you understand which form your text/software library supports.

Quaternions are a form of hyper-complex number where instead of a single real and single imaginary component, we have one real and THREE imaginary components (i, j & k). Rules for these imaginary components are:

$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$ (Eqn. 4)

$\mathbf{ij} = \mathbf{k} = -\mathbf{ji}$ (Eqn. 5)

$\mathbf{jk} = \mathbf{i} = -\mathbf{kj}$ (Eqn. 6)

$\mathbf{ki} = \mathbf{j} = -\mathbf{ik}$ (Eqn. 7)

Two quaternions, p and q, are equal to one another only if the individual components are equal. You add two quaternions by adding the individual components. If

$p = p0 + ip1 + jp2 + kp3$; and (Eqn. 8)

$q = q0 + iq1 + jq2 + kq3$ (Eqn. 9)

Then

$p + q = (p0 + q0) + \mathbf{i}(p1 + q1) + \mathbf{j}(p2 + q2) + \mathbf{k}(p3 + q3)$ (Eqn. 10)

The addition operation commutes. That is p + q = q + p. Multiplication of a quaternion by a scalar real number is trivial, just multiply each of the four components by the scalar. Multiplication of two quaternions is NOT so trivial:

$pq = p0q0 – \mathbf{p}.\mathbf{q} + p0\mathbf{q} + q0\mathbf{p} + \mathbf{p} \times \mathbf{q}$ (Eqn. 11)

Multiplying one quaternion by another quaternion results in a third quaternion. Notice that the 1st two components (p0q0 – p.**q**) makes up the scalar portion of the result, and the last three (p0**q** + q0**p** + **p** x **q**) comprise the vector portion. The quaternion product operation is not commutative pq ≠ qp. Order matters. Multiplication of two quaternions includes scalar, cross product and dot product terms. Unless you are writing your own quaternion library, you are likely never to use the expression above. Instead, you will use a function that does the quaternion multiplication for you.

The complex conjugate of

$q = q0 + \mathbf{i}q1 + \mathbf{j}q2 + \mathbf{k}q3$ is $q^* = q0 – \mathbf{i}q1 – \mathbf{j}q2 – \mathbf{k}q3$ (Eqn. 12)

Related to this, we have

$(pq)^* = q^*p^*$ (Eqn. 13)

$q + q^* = 2q0$ (Eqn. 14)

$q – 1 = q^*$ for any unit quaternion (Eqn. 15)

Eqn. 15 is interesting. If you think of a quaternion as a rotation operator, it says you can reverse the sense of rotation by inverting the axis of rotation. Given our usual standard of using the Right Hand Rule to describe the polarity of rotations, this makes perfect sense. Reversing the direction of the axis is equivalent to reversing the direction of rotation.

Another interesting take on the above is that rotation quaternions are not unique:

$q = –q$ (Eqn. 16)

Any rotation quaternion can be multiplied by –1 and still result in the same rotation! That is because we reversed both the angle AND the axis of rotation, which then cancel each other. It is conventional, therefore, to remove the ambiguity by negating a rotation quaternion if its scalar component is negative.

At this point, you are surely wondering why in the world you might, or might not, choose to use quaternions instead of rotation matrices. Here is a brief summary of the pros and cons.

Table 26. Pros & Cons of the Different Orientation Representations

Topic	Quaternion	Rotation Matrix
Storage	Requires **16 bytes** of storage in single precision floating point (4 elements at 4 bytes each)	Requires **36 bytes** of storage (9 elements at 4 bytes each)
Computation (for 2 sequential rotations)	4 elements each requiring 4 multiplies and 3 additions = **28 operations**	9 elements, each requiring 3 multiplies and 2 additions = **45 operations**
Vector rotation	Rotating a vector by pre- and post-multiplication of quaternion requires **52 operations**	Rotating a vector via rotation matrix requires **15 operations** (3 elements each requiring 3 multiplies and 2 additions)
Discontinuities	Generally, we force the scalar part of the quaternion to be positive, which can cause a discontinuity in the rotation axis (it flips).	None
Ease of Understanding	Generally, takes a lot of study to understand the details	Easily understood by most engineers
Conversion	From rotation matrix = $\begin{array}{\|c\|c\|c\|}\hline m11 & m12 & m13 \\ \hline m21 & m22 & m23 \\ \hline m31 & m32 & m33 \\ \hline \end{array}$ We have q0 = 0.5 sqrt(m11 + m22 + m33 + 1) q1 = (m32 – m23) / (4q0) q2 = (m13 – m31) / (4q0) q3 = (m21 – m12) / (4q0) (Eqn. 17)	RM = $\begin{array}{\|c\|c\|c\|}\hline 2q02 – 1 + 2q12 & 2q1q2 – 2q0q3 & 2q1q3 + 2q0q2 \\ \hline 2q1q2 + 2q0q3 & 2q02 – 1 + 2q22 & 2q2q3 – 2q0q1 \\ \hline 2q1q3 – 2q0q2 & 2q2q3 + 2q0q1 & 2q02 –1 + 2q32 \\ \hline \end{array}$ (Eqn. 18)

Equations 17 and 18 are consistent with regards to direction of rotation. If instead of rotating a vector in a fixed frame of reference, you rotate the frame of reference itself, you will need to use the transpose of Eqn. 18 and invert q1, q2 and q3 in Eqn. 17.

Returning to the quaternion rotation operator **W** = qVq* , note that **V** needs to be expressed as a quaternion of the form [0, vx, vy, vz], and the multiplications are quaternion multiplies as defined in Eqn. 11. q* is the complex conjugate defined in Eqn. 12.

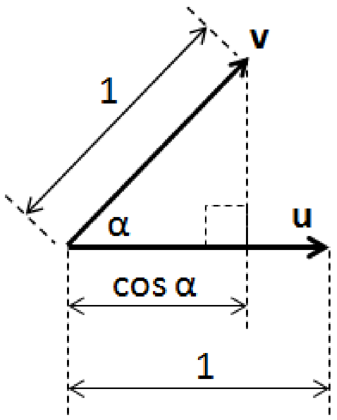
If you do a lot of graphics or sensor fusion work, you will probably find yourself constantly switching between the various representations we have considered. You will find it useful to remember a couple of identities from your high school geometry course:

10.2.1.1 The Dot Product

u. **v** = | **u** | | **v** | cos α (Eqn. 19)

If both **u** and **v** are unit vectors, then:

u. **v** = cos α (Eqn. 20)



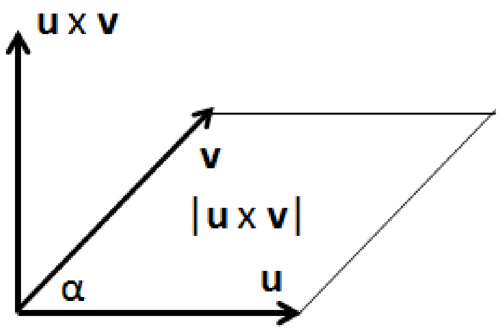
10.2.1.2 The cross product

u x **v** = | **u** | | **v** | sin α **n** (Eqn. 21)

where **n** is a unit vector perpendicular to the plane containing **u** and **v** (the polarity of **n** follows the right hand rule).

If both **u** and **v** are unit vectors, then:

n = **u** x **v** / (sin α) (Eqn. 22)



If you have been paying attention, you will see that α is the rotation of u into v about the axis of rotation defined by u x v. See! It is simple! Axis and angle!

10.2.2 References

[1] Quaternions and Rotation Sequences, Jack B. Kuipers, Princeton University Press, 1999

[2] Euler Angles from the Wolfram Demonstrations Project by Frederick W. Strauch

[3] Diversified Redundancy in the Measurement of Euler Angles Using Accelerometers and Magnetometers, Chirag Jagadish and Bor-Chin Chang, Proceedings of the 46th IEEE Conference on Decision and Control, Dec. 2007

[4] “Euler Angles” at Wikipedia

11. Theory: hard and soft iron magnetic compensation

This section is a variation on a posting (Hard & Soft Iron Magnetic Compensation Explained) which initially appeared on the Freescale (now NXP) Embedded Beat blog site in 2011.

This section explores issues that you may encounter when using any magnetic sensor in consumer applications. To keep things simple, let us consider the case where you are integrating a magnetic sensor into a smart phone. Nominally, you would like to use your magnetic sensor to implement compass and navigations features. In a pristine environment, free from interference, we could use measurements taken directly from our sensor. The real world is not that simple.

11.1 Distortion of the magnetic field due to the presence of soft iron.

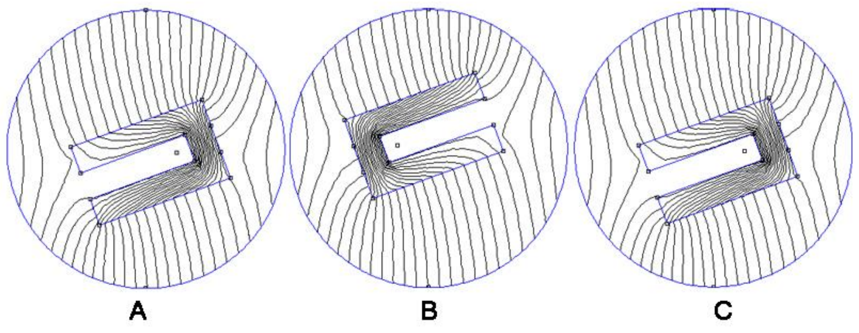


Fig 56. Soft iron distortion in a uniform field

Soft iron, you say? What's that? Think steel. Think EMI shields, screws and battery contacts. To illustrate the point, we performed a finite elements simulation of a U-shaped piece of steel sitting in a uniform magnetic field. Fig 56A shows how the magnetic field (which would otherwise be shown as vertical lines) is distorted by the presence of our "U-bar". Steel provides a "lower resistance" path to the magnetic field than does the surrounding air. So it is natural for the field to be diverted.

Fig 56B takes that same U-bar and rotates it exactly 180 degrees in the same ambient field. You can see similarities in the field distortion. We can see just how similar Fig 56A and Fig 56B are by taking Fig 56B and flipping it, first about one axis and then the other, to obtain Fig 56C, which **is identical in form to Fig 56A.**

This makes a lot of sense when you realize that *from the steel's perspective*, Fig 56A and Fig 56B are identical except for the polarity of the ambient magnetic field. We *should* get symmetrical results.

More importantly, we are going to be able to use this simple observation to remove the distortion caused by soft iron from our measurement of the ambient magnetic field.

To see how, let us take this same U-bar and rotate it in 20 degree increments in the same field. At the same time, let us measure and plot the magnetic field at the "dot" you see nestled near the base of the "U". It's important to note that this point is fixed relative to the disturbing metal. They rotate together.

The symmetry seen above continues to hold as we rotate our soft iron. The field distortion at each angle of rotation matches (after the "flips" noted above) the distortion seen at that angle + 180 degrees. More importantly, the field magnitude measured at each angle matches the field magnitude measured at that angle + 180 degrees.

If we plot the x/y sensor readings for all of our points, we will get an ellipse, Fig 57. This is a function of the basic physics and always holds true, regardless of the sensor type used to make the measurement.

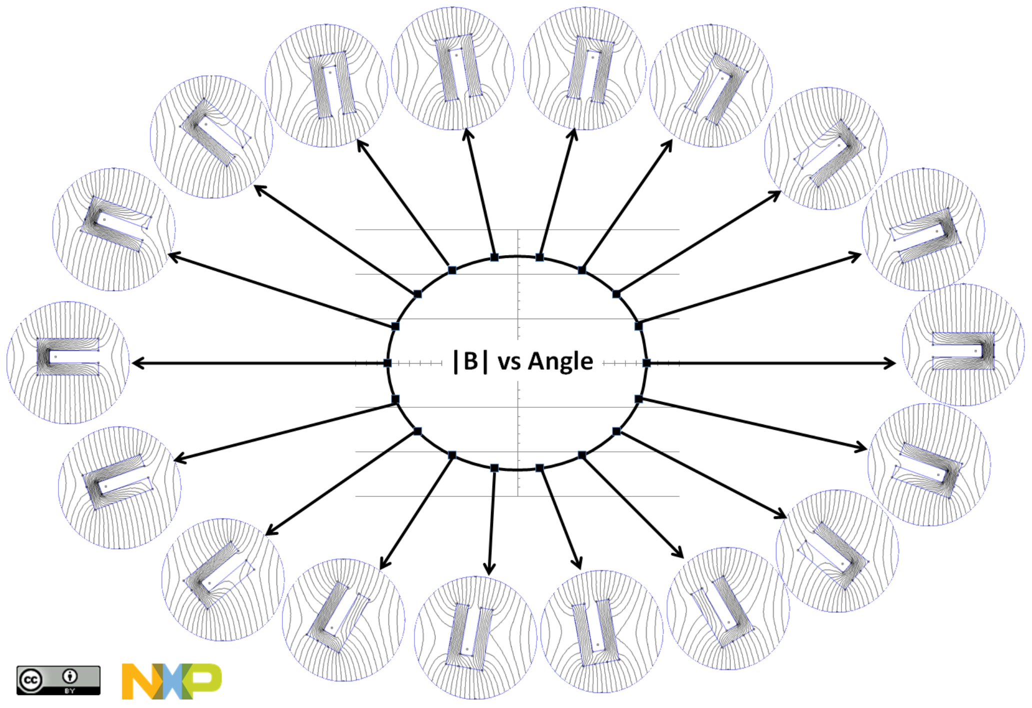


Fig 57. Soft iron fixed relative to measurement point

Both are rotated together in the ambient field.

If there were no soft iron present and we simply rotated our sensor, the ellipse would collapse into a simple circle. Since the field remains the same regardless of the angle of measurement, this must be the case. So we see that the effect of soft iron is to distort a circle whose radius is equal to the magnitude of the ambient magnetic field into an ellipse.

This result can be extended to three dimensions. Measurements taken while rotating a sensor in free space undisturbed by hard or soft iron can be visualized as a sphere with fixed radius equal to the magnitude of the ambient magnetic field. Adding soft iron to the mix will distort that sphere into a 3D ellipsoid. See Fig 58.

The equation for a 3D sphere is:

$$x^2 + y^2 + z^2 = r^2.$$

In matrix form, if we have X = [x y z], then

$$XX^T = r^2.$$

For an ellipsoid, it is:

$$x^2/a + y^2/b + z^2/c = 1.$$

In matrix form this is

$$XAX^T = 1, \text{ where}$$

$$A = \begin{pmatrix} 1/a & 0 & 0 \\ 0 & 1/b & 0 \\ 0 & 0 & 1/c \end{pmatrix}$$

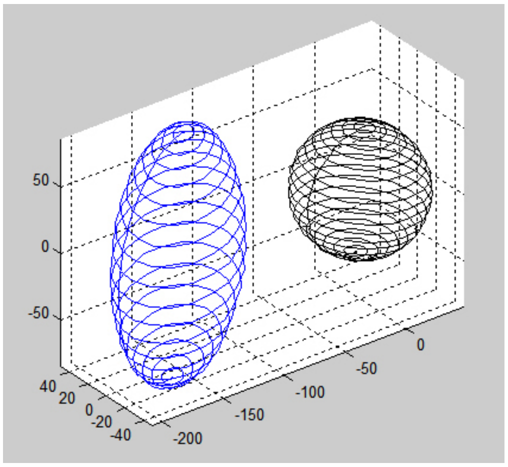


Fig 58. Distorted (left) and ideal (right) magnetic fields

You can see that the equation of the sphere can be linearly derived from that of the ellipsoid and vice versa. If we take a representative set of samples on the surface of our ellipsoid, we can, through a variety of methods, determine the reverse mapping from distorted to undistorted magnetic field readings. Essentially, we are curve fitting in three dimensions. Now on to…

11.2 Distortion of the magnetic field due to the presence of hard iron

“Hard iron” is just a physicist’s way of saying “permanent magnet”. “Why,” you might ask, “am I worried about permanent magnets?” Well, that smart phone we mentioned must have a speaker. And speakers have magnets. And a lot of phone holsters have magnets to secure the device. So it turns out that yes, we DO have to deal with them. The good news is that, compared to soft iron effects, compensating for hard-iron offsets is relatively simple. If a magnet is fixed in location and orientation with respect to our sensor, then there is an additional constant field value added to the value that would otherwise be measured. If only soft-iron effects are present, the ellipsoid mentioned above should be centered at [x,y,z] = [0,0,0]. A permanent magnet fixed relative to the measurement point simply adds an offset to the origin of the ellipsoid. If we have a large enough data set, we can determine that offset as

$$\text{hard-iron offset} = [x_{max} + x_{min},\ y_{max} + y_{min},\ z_{max} + z_{min}]/2$$

This technique will NOT work for magnets that move with respect to the sensor. The magnet on the phone holster flap cannot be permanently canceled out. But that is good news! A sudden shift in offset/magnitude of our calculated field probably implies that the phone has been inserted or removed from its holster. That can be a useful thing to know.

11.2.1 Implications

The techniques discussed generally employ some form of curve fitting, which raises subtle issues that do not get discussed much: How many data points do we need in our constellation of sample points? How often does that constellation need to be updated? How do we decide to add or drop points to/from the constellation? What should we do when a sudden change in ambient field magnitude is detected? What numerical method(s) should be used to calculate the trim parameters? How do you deal with uncorrelated magnetic disturbances that occur around us every day? How do you deal with field variations with temperature? The answers to these questions make up much of the “secret sauce” used by various vendors in their calibration algorithms.

Earth ambient magnetic field as a function of zip code can be obtained from the online National Geophysical Data Center magnetic field calculator. But for some applications, you may not even care what the local value is. The important point to take from the discussion above is that we are leveraging the symmetry of the data set to drive a solution. Compass orientation is determined by ratios of the three dimensions calculated. If those values are off from the expected values by a multiplicative constant, the ratios still hold.

See Application Note AN5019, *Magnetic Calibration Algorithms*, to learn about the specific magnetic calibration algorithms used within the NXP Sensor Fusion Library.

We have also listed a few papers in the references, Section 12. Although often intense, do not let the math scare you. Almost all have the basic assumptions discussed above at the heart of their approach. Some solve the problem only in the X/Y plane, others address all three dimensions. Tradeoffs include program and variable memory space, CPU cycles required and ease of implementation.

A basic requirement of these approaches is that the sensor output data vary linearly with magnetic field. If your sensor is nonlinear, correction factors must be applied prior to applying the techniques discussed.

12. References

[1] C. Verplaetse, “Inertial proprioceptive devices: Self-motion-sensing toys and tools”, IBM Systems Journal, 1996, Vol. 35, pp. 639–650

[2] Magnetometer Autocalibration Leveraging Measurement Locus Constraints, Demoz Gebre-Egziabher, Journal of Aircraft, Vol. 44, No. 4, July–August 2007

[3] National Geophysical Data Center magnetic field calculator at http://www.ngdc.noaa.gov/geomagmodels/IGRFWMM.jsp.

[4] A Geometric Approach to Strapdown Magnetometer Calibration in Sensor Frame, Vasconcelos, Elkaim, Oliveira & Cardeira

[5] A Nonlinear, Two-Step Estimation Algorithm for Calibrating Solid-State Strapdown Magnetometer, Gebre-Egziabher, Elkaim, Power & Parkinson.

[6] Iterative calibration method for inertial and magnetic sensors, Dec. 2009, Dorveaux, Vissiere, Martin & Petit

[7] A Research of an Improved Ellipse Method in Magnetoresistive Sensors Error Compensation, Aug. 2009, Lian-yan, Qing & Wen-yuan

[8] Finite Element Method Magnetics, FEMM 4.2.

[9] Characterization of Various IMU Error Sources and the Effect on Navigation performance, Flenniken, Wall and Bevly, 2005, 18th International Technical Meeting, Institute of Navigation

[10] U.S. Standard Atmosphere, 1976, available at http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770009539.pdf

13. Legal information

13.1 Definitions

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

13.2 Disclaimers

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp/com/SalesTermsandConditions.

13.3 Trademarks

NXP, the NXP logo, Freescale and the Freescale logo are trademarks of NXP B.V. All other product or service names are the property of their respective owners.

14. Index

15. List of figures

16. List of tables

17. Contents

Date of release: 30 May 2017

Document identifier: NSFK_Prod_UG

Date of release: 30 May 2017

Document identifier: NSFK_Prod_UG