

---

# **OpenIMU Documentation**

**Aceinna Engineering**

**May 24, 2021**

---

## Table of contents

---

<b>I</b>	<b>OpenIMU</b>	<b>2</b>
1	Overview	3
2	WARNING!!!! Before You Start Development	8
3	Tools	12
4	Ready-to-Use Applications	29
5	Tutorial - What The User Needs to Know to Build The First Application	46
6	OpenIMU Software Overview	68
7	Algorithm Simulation System	121
8	Python Serial Driver	122
<b>II</b>	<b>Products</b>	<b>123</b>
9	OpenIMU300ZI - <i>EZ Embed</i> Industrial Module	124
10	OpenIMU300RI - Rugged Industrial CAN Module	139
11	OpenIMU330BI - Triple Redundant, 1.5 °/Hr, SMT Module	156
12	OpenIMU335RI - Triple-Redundant Rugged Industrial CAN Module	178
<b>III</b>	<b>Dev Support Algorithms</b>	<b>189</b>
13	OpenIMU Hardware/Software Interface	190
14	EKF Algorithms	194
15	Magnetic Sensor Algorithms	222

<b>IV</b>	<b>Miscellaneous</b>	<b>223</b>
<b>16</b>	<b>C-Code Serial Driver</b>	<b>224</b>
	<b>Index</b>	<b>225</b>



OpenIMU is a precisely calibrated open source Inertial Measurement Unit platform. Users are able to quickly develop and deploy custom navigation/localization algorithms and custom sensor integrations on top of the OpenIMU platform. OpenIMU also has pre-built drivers in Python as well as a developer website - Aceinna Navigation Studio (ANS). These tools make logging and plotting data, including custom data structures and packets very simple.

**Social:** [Twitter](#) | [Medium](#)



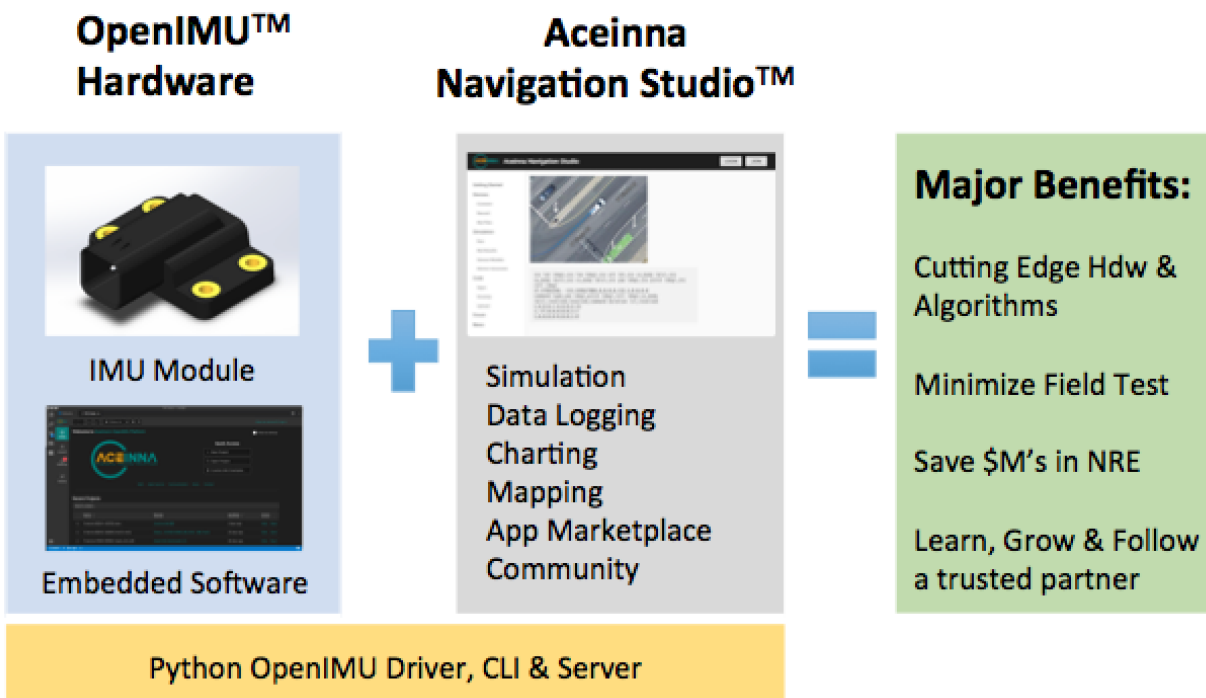
# **Part I**

# **OpenIMU**

# CHAPTER 1

## Overview

OpenIMU is a precisely calibrated, open-source Inertial Measurement Unit platform for the development of navigation and localization algorithms. A free Visual Studio Code (VSCode) extension is installed which contains all the software and tools necessary to create and deploy custom embedded sensor apps using OpenIMU. Visual Studio Code is the recommended IDE and the extension configures VS Code to include easy access to compilation, code download, JTAG debug, IMU data logging as well as OpenIMU platform updates and news. A developer website called Aceinna Navigation Studio (ANS) includes additional support tools including a GUI for controlling, plotting and managing data files logged by your Custom IMU.



The OpenIMU and ANS platform and tool-chain are supported on all three Major OS cross-development platform:

- Windows 7 or 10
- MAC OS 10
- Ubuntu 14.0 or later

---

**Note:** Contributions to the public repositories related to this project are welcomed. Please submit a pull request.

---

The following pages cover:

- What is OpenIMU
- What is the Acienna Navigation Studio
- Who is using OpenIMU and the Acienna Navigation Studio

## 1.1 What is OpenIMU?

- OpenIMU is an open software platform for development of high-performance navigation and localization applications on top of a family of low-drift pre-calibrated Inertial Measurement Units (IMU).
- OpenIMU hardware consists of a 3-axis rate sensor (gyro), 3-axis accelerometer platform, and 3-axis magnetometer module.
- The module contains a low-power embedded ARM Cortex-M4 CPU with floating-point math support. Extra IO and Ports make connection of external peripherals such as GPS, Odometer, and other more advanced sensors possible.
- The OpenIMU hardware comes in different form-factors including:

### Hardware Configurations

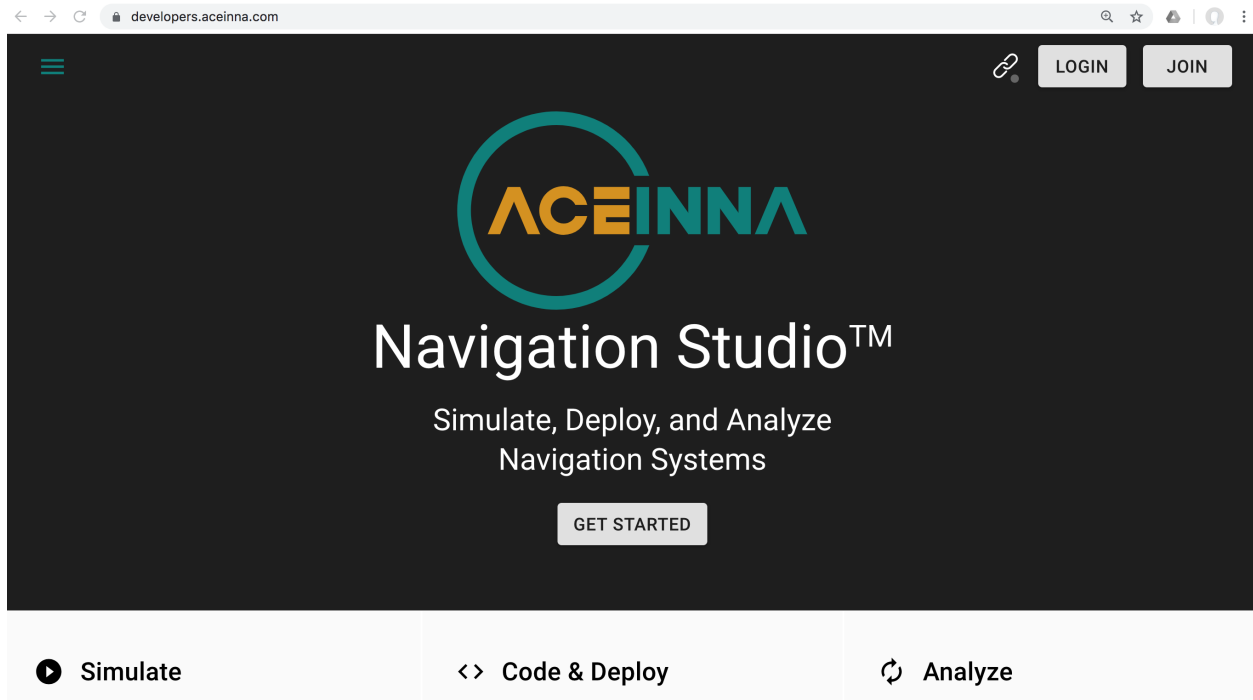
Type	Part Number	Hardware Features
<i>EZ</i>	OpenIMU300ZI	Easy to Embed 3-5V UART/SPI Industrial IMU Module
<i>CAN</i>	OpenIMU300RI	Rugged, Waterproof 5-32V CAN/RS232 Industrial Module
<i>SMT</i>	OpenIMU330BI	Triple Redundant, SMT, 2°/Hr IMU

### Open-Source Embedded Software

- OpenIMU hardware runs an open-source stack written on top of standard ARM Cortex libraries.
- OpenIMU300 use FreeRTOS while OpenIMU330 uses a simple real-time scheduler
- The open-source stack includes EKF (Extended Kalman Filter) algorithms that can be used directly or customized for application specific use.
- The overall system loop is typically configured to run at 800Hz ensuring high quality aliasing-free measurements for processing.
- Also included in the OpenIMU embedded software platform are drivers for various GPS receivers, customizable SPI, CAN, and UART messaging, and customizable settings that can be adjusted run-time and/or permanently.
- A number of predefined settings are defined for baud rate, output date rate, sensor filter settings, and XYZ axis transformations.
- The Core OpenIMU embedded software consists of the following:
  - FreeRTOS
  - Extended Kalman Filter Algorithms

- High-Speed Deterministic Sampling
- Messaging
- GPS Drivers
- Accurate Time Service
- Sensor Filtering
- Settings Module for Dynamic and Permanent Unit Configuration

## 1.2 What is Aceinna Navigation Studio?



- The Aceinna Navigation Studio (<https://developers.aceinna.com>) is a navigation system developer’s website and web-platform.
- It consists of a graphical user interface to control and configure OpenIMU units.
- Using a JSON configuration file (“openimu.json”), the graphical user interface can be customized for user specific messaging and settings without any additional coding. This aligns the embedded code with both the Python device server and the GUI pages available on ANS (<https://developers.aceinna.com>).
- Online tools include graphing, mapping, logging, and simulation.
- User Forum is available at (<https://forum.aceinna.com>).

### Python & the Acienna Navigation Studio

The Acienna Navigation Studio (ANS) requires Python to operate. If the user has not installed Python, it can be installed from <https://www.python.org/downloads/>. Download and install the latest version.

An open-source Python driver for OpenIMU is available and required. The Python driver can be used directly from the terminal to load, log, and test your application. The driver leverages the PySerial library to connect to an OpenIMU

of a serial connection. The python script supports configuring units, firmware updates (JTAG is faster for debugging), and local data logging.

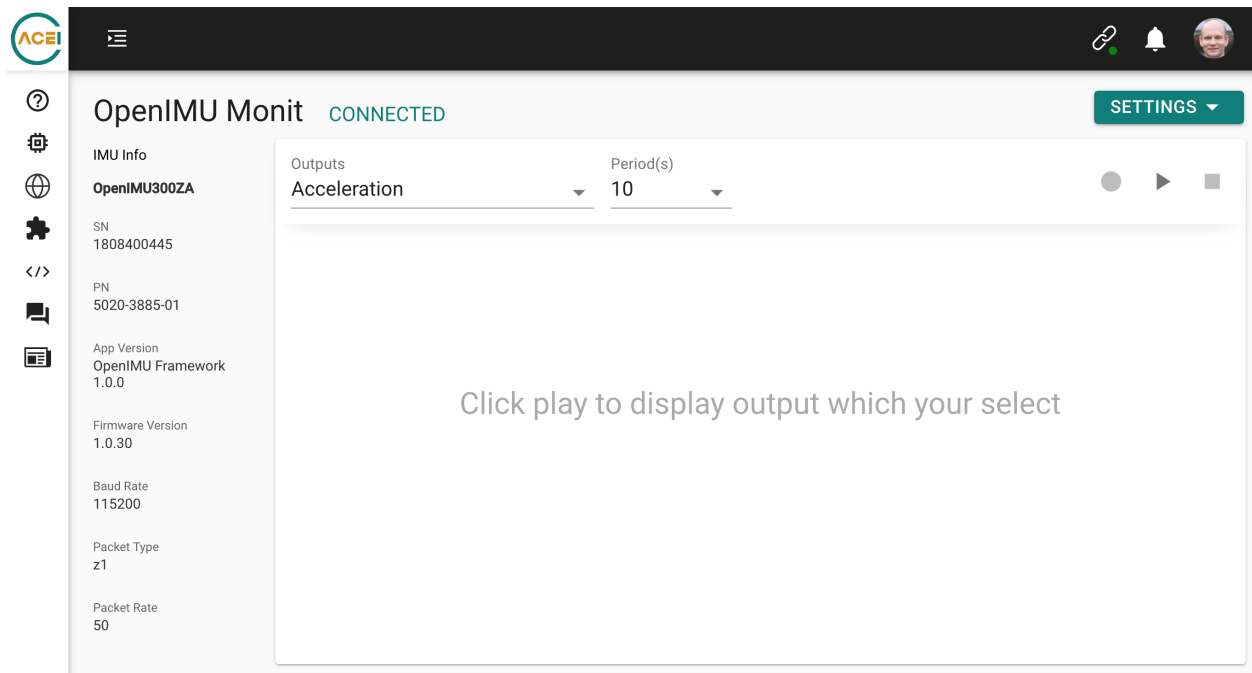
In addition, the open-source Python driver can acts as a server connecting the OpenIMU hardware with our ANS developer platform for a GUI experience, cloud data storage and retrieval, as well as stored file charting/plotting tools.

The Aceinna VS Code extension ensures a python environment automatically. The OpenIMU python code can be installed independently by cloning the repository <https://github.com/python-openimu> or using pip as shown below.

```
pip install openimu
```

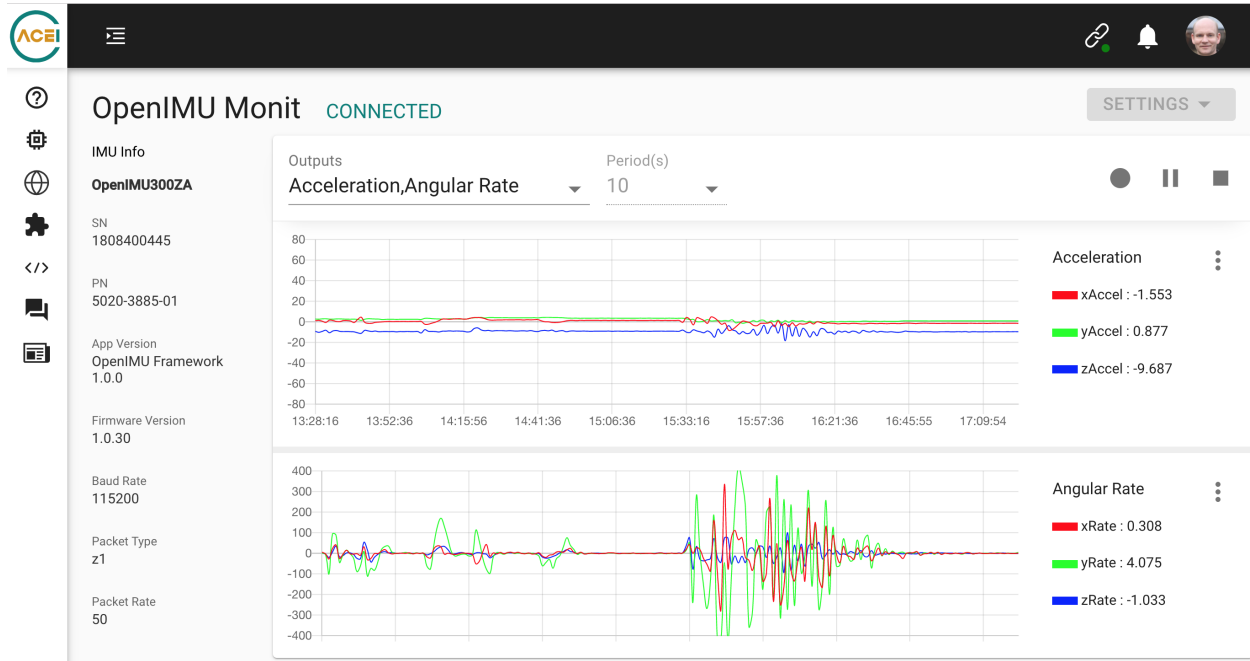
## Connection

- Connection Status is shown on the link symbol at the top right hand side of the page.
- Device information is exposed on the main IMU page. (<https://developers.aceinna.com/devices/record-next>)
- The baseline OpenIMU firmware provides a set of “standard settings” such as baud rate, output data rate, and more.
- Custom options are added by adding additional options to “UserConfiguration” in *both* the OpenIMU embedded C code as well as the the openimu.json file which provides a summary of the descriptions and potential values for the UI.



## Graphing

Use the play, record, and stop buttons to log data.



## File Retrieval

Logged files are retrieved on the My Files page which opens up a zoomable graph view. *Requires Login*

**My Log Files**

IMU-300		INS-1000	RTK LOGIN LOGS	RTK ERROR LOGS	RTK ROVER POSITION			
SN	File Name	Part Number	Serial Number	App Version	Packet Type	Packet Rate	Create Date	
1	parkinglot_2019_08_16_13_21_51.csv	5020-3885-01	OpenIMU300ZA		e2	100	2019-08-16 13:24:59	
2	filterTest_2019_08_03_14_59_25.csv	5020-3885-01	OpenIMU300ZA		z1	100	2019-08-03 14:59:43	
3	mike2.csv	OpenIMU300ZA	OpenIMU300ZA	1.0.0	z1	100	2019-03-13 16:37:43	
4	quicktest.csv	OpenIMU300ZA	OpenIMU300ZA	1.0.0	a1	50	2018-12-02 23:49:46	

## 1.3 Who is using it?

The OpenIMU project is recommended for **autonomous system** developers with challenging navigation and localization requirements. The system is being used by several autonomous driving teams globally.

## CHAPTER 2

---

### WARNING!!!! Before You Start Development

---

#### Contents

- *Save unit image:*
- *Recover unit image:*

**Before You start developing it is recommended to read whole unit image and save it to binary file to be able to recover unit if something unexpected happened**

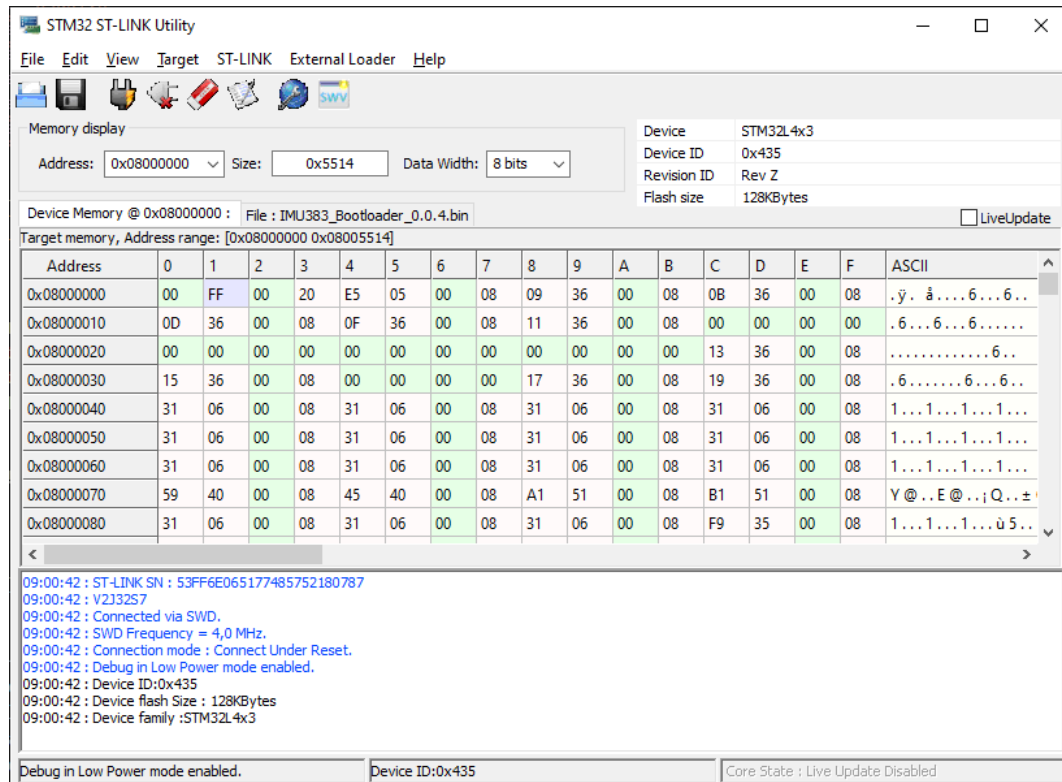
Unit image consists of:

1. Bootloader
2. Original (factory) application image
3. Calibration and Configuration partitions.

**If bootloader or calibration tables are damaged - unit will not work properly!!!.**

### 2.1 Save unit image:

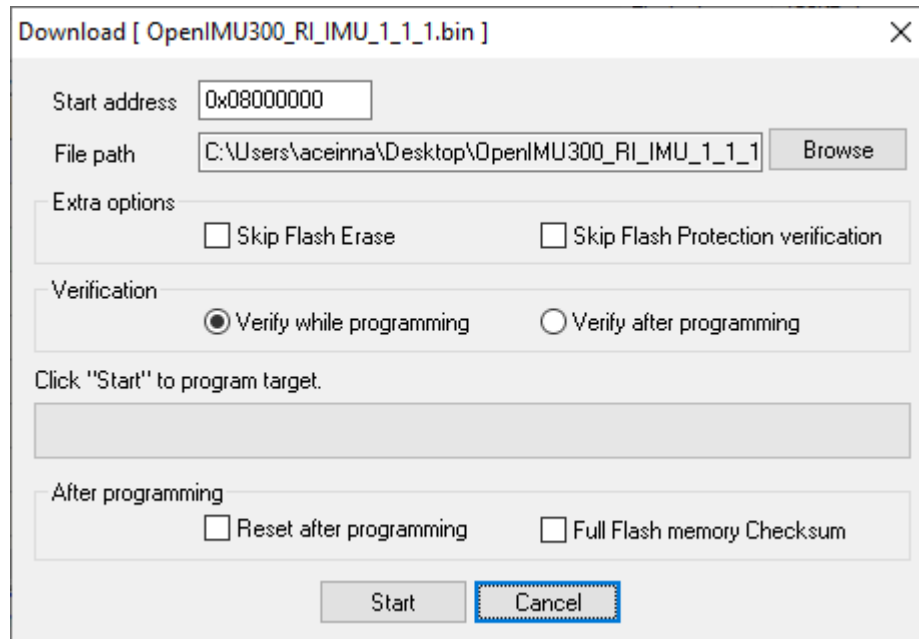
1. Install ST-Link Utility from here: <https://www.st.com/en/development-tools/stsw-link004.html>
2. Connect ST-Link debugger to OpenIMU Evaluation Kit. and to PC
3. Power On Evaluation Kit
4. Start ST-Link utility on your PC.
5. Click Device->Connect.
6. Enter value **0x80000 for OpenIMU300** and **0x20000 for OpenIMU330** into Size box and hit enter
7. Click File->SaveAs and save image to well known location. For OpenIMU300 image size should be 512K bytes. For OpenIMU330 image size should be 128 KBytes.



## 2.2 Recover unit image:

1. Connect ST-Link debugger to OpenIMU Evaluation Kit. and to PC
2. Power On Evaluation Kit
3. Start ST-Link utility on your PC.
4. Click Device->Connect.
5. Click File->open and open previously saved file.
6. Click Target->Program&Verify.
7. Make sure that **Start address is 0x08000000** and click Start.





8. After reprogramming of **OpenIMU300** unit (RI or ZI) perform write protection of sectors 0 and 2
9. After reprogramming of **OpenIMU330BI** unit perform write protection of last 6 sectors (58 to 63)

**Option Bytes** [X]

Read Out Protection: Level 0 [v]  
 BOR Level: F Level 0 [v] R [v]

User configuration option byte

<input checked="" type="checkbox"/> IWDG_STOP	<input checked="" type="checkbox"/> IWDG_STDBY	<input type="checkbox"/> nBoot0	<input checked="" type="checkbox"/> nBOOT0
<input checked="" type="checkbox"/> WWDG_SW	<input type="checkbox"/> IWDG_ULP	<input checked="" type="checkbox"/> nBoot1	<input type="checkbox"/> BOOT1
<input type="checkbox"/> nSRAM_Parity	<input type="checkbox"/> FZ_IWDG_STOP	<input type="checkbox"/> nDBOOT	<input type="checkbox"/> nBFB2
<input checked="" type="checkbox"/> SRAM2_RST	<input type="checkbox"/> FZ_IWDG_STDBY	<input type="checkbox"/> nDBANK	<input type="checkbox"/> nBOOT_SEL
<input checked="" type="checkbox"/> SRAM2_PE	<input checked="" type="checkbox"/> PCROP_RDP	<input checked="" type="checkbox"/> DB1M	<input type="checkbox"/> DUALBANK
<input checked="" type="checkbox"/> nRST_SHDW	<input type="checkbox"/> nBoot0_SW_Cfg	<input type="checkbox"/> IRHEN	<input type="checkbox"/> BOREN
<input checked="" type="checkbox"/> nRST_STOP	<input checked="" type="checkbox"/> nSWBOOT0	<input checked="" type="checkbox"/> WDG_SW	
<input checked="" type="checkbox"/> nRST_STDBY	<input type="checkbox"/> VDDA_Monitor	<input type="checkbox"/> SDADC12_VDD_Monitor	

NRST\_MODE: [v]

Security option bytes

SEC\_SIZE: 0x00 SEC\_SIZE2: 0x00 ☐ BOOT\_LOCK

Boot address option bytes

BOOT\_ADD0 (H): [ ] Boot from (H): [ ]

BOOT\_ADD1 (H): [ ] Boot from (H): [ ]

User data storage option bytes

Data 0 (H): [ ] Data 1 (H): [ ]

Flash sectors protection

Write Protection Read/Write Protection (PCROP)

Page	Start address	Size	Protection
<input type="checkbox"/> Page 55	0x0801B800	2 K	No Protection
<input type="checkbox"/> Page 56	0x0801C000	2 K	No Protection
<input type="checkbox"/> Page 57	0x0801C800	2 K	No Protection
<input checked="" type="checkbox"/> Page 58	0x0801D000	2 K	Write Protection
<input checked="" type="checkbox"/> Page 59	0x0801D800	2 K	Write Protection
<input checked="" type="checkbox"/> Page 60	0x0801E000	2 K	Write Protection
<input checked="" type="checkbox"/> Page 61	0x0801E800	2 K	Write Protection
<input checked="" type="checkbox"/> Page 62	0x0801F000	2 K	Write Protection
<input checked="" type="checkbox"/> Page 63	0x0801F800	2 K	Write Protection

Unselect all Select all

Apply Cancel

This section reviews more detail on various Tools available for OpenIMU development environment:

### 3.1 PC Tools Installation

#### Platforms - Computers with the following Operating Systems

- Windows 10 or 7
- Ubuntu version 14.0 or later
- MAC OS

#### Visual Studio Code

Visual Studio Code - can be downloaded from here: <https://code.visualstudio.com>

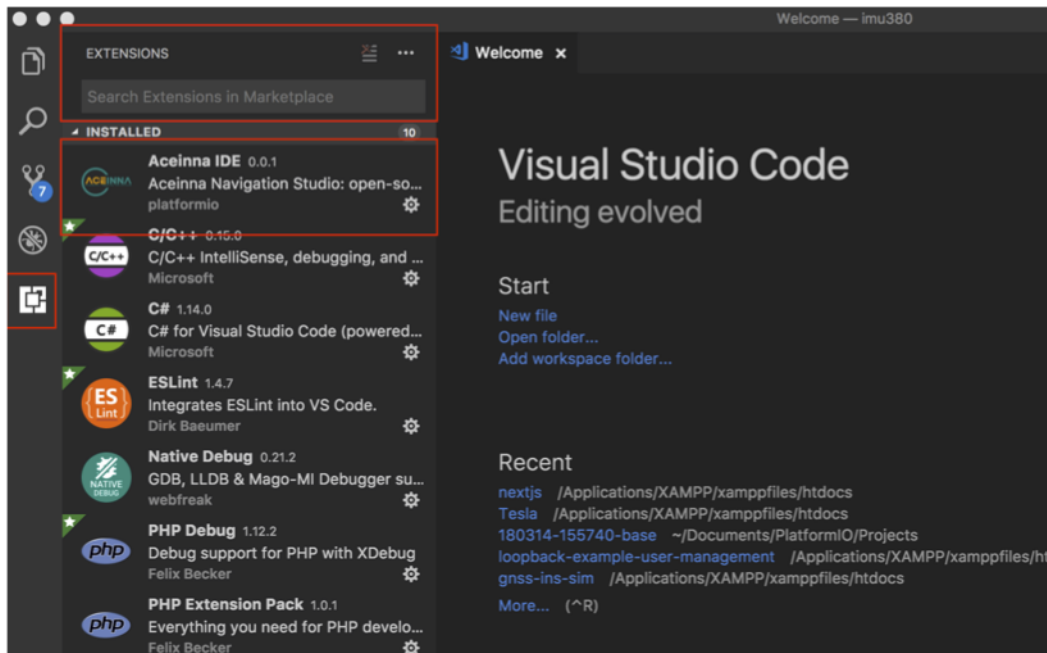
#### ST-LINK Debugger Driver

- *MacOS* - ST-LINK drivers are automatically installed for MAC OS.
- *Windows* - ST-LINK drivers should be also installed automatically. But in case if it was not - ST-LINK V2 driver can be manually installed for Windows. The Windows driver is downloaded from the following page link: <http://www.st.com/en/development-tools/st-link-v2.html>
- *Ubuntu* - please see step 5.

#### Installation of OpenIMU development platform

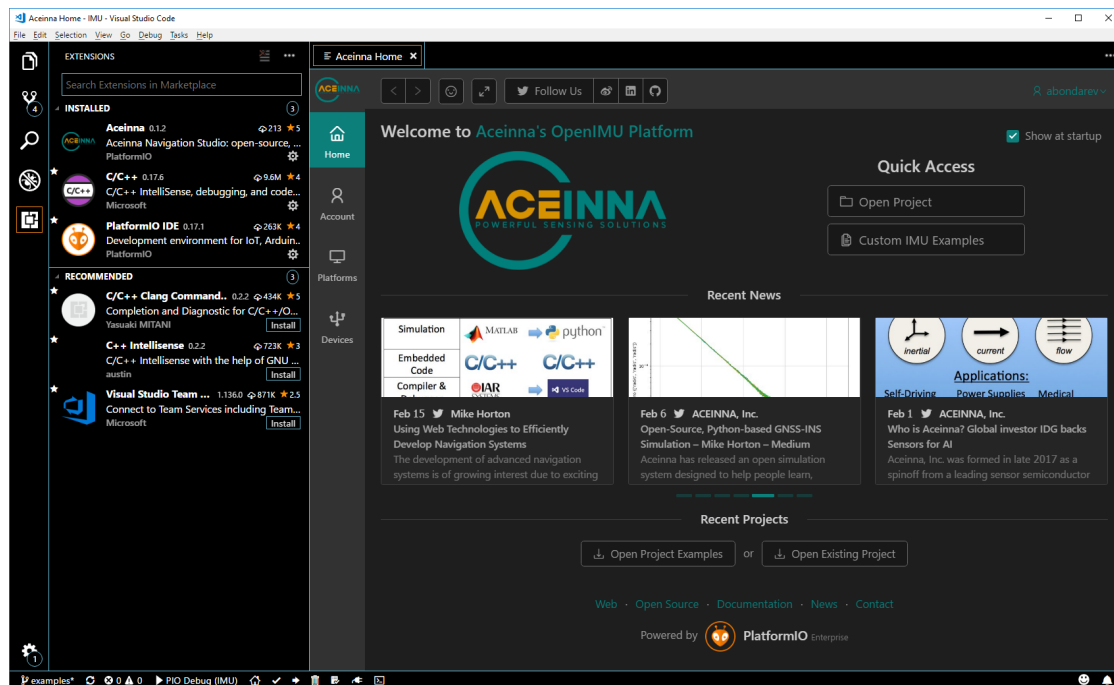
To install OpenIMU development platform:

1. Start Visual Studio Code.
2. On leftmost toolbar find “Extensions” icon and click on it.
3. In the text box “Search extensions on Marketplace” type “Aceinna” and hit enter
4. Install Aceinna Extension and Follow prompts.



### First steps

After installation of “Aceinna” extension click on “Home” icon at the bottom of the screen. It will bring up Aceinna OpenIMU platform homepage. Click on “Custom IMU examples”, chose desired example and click “Import”.



The required example will be imported into working directory in folder:

C:\Users\<username>\Documents\platformio\Projects\ProjectName

Now you can edit, build and test the project. All your changes will remain in the above-mentioned directory and subdirectories. Next time when you return to development - open Aceinna “Home” page

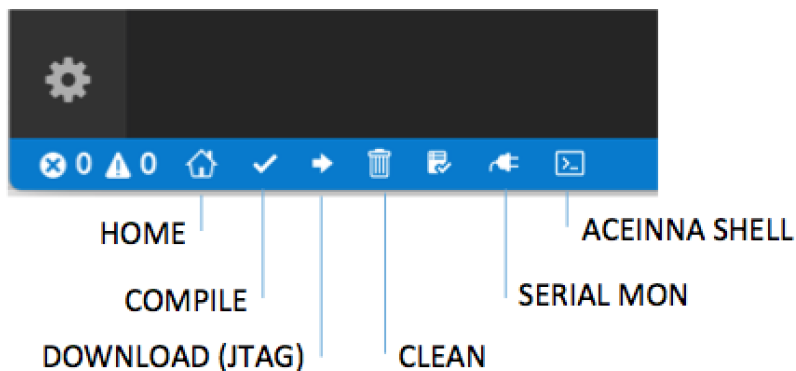
and click “Open Project”, choose “Projects” and select required project from the list.

The source tree of imported project tree has the following structure:

```
project directory -|
|
|--- .pio --|
|               |-- build --|
|               |               |-- board-|
|               |               |-- binary image_
→ (firmware.bin) |               |
|               |               |-- elf image (firmware.
→elf)           |               |
|               |               .
|               |               .
|               |               .
|               |-- libdeps -|
|               |               |-- board-| Library dependencies
|               |               |
|               |               |--library1 src tree
|               |               |
|               |               |--library2 src tree
|               |               |
|               |               |--library3 src tree
|               |               |
|               |               .
|               |               .
|
|--include (optional user include files)
|--lib (optional user library directory tree)
|--src (user source files tree)
```

### Compile and JTAG Code Loading

Once you have imported an example project, a good first step is to compile and download this application using your ST-LINK. At the bottom of the VS Code window is the shortcut toolbar shown below. To load an application to the OpenIMU with JTAG, simply click the Install/Download button while the ST-LINK is connected to your EVB.



The OpenIMU development environment uses PlatformIO's powerful open-source builder and IDE. This on-line manual focuses on on OpenIMU specific information, and it does not attempt to fully discuss all of the IDE's powerful features in depth. For more information on PlatformIO builder and IDE features include command line interface, scripting and more please see the [PlatformIO](#)

## 5. ST-LINK Install for Ubuntu (Manual Version)

Go to <https://github.com/texane/stlink> and read instructions carefully.

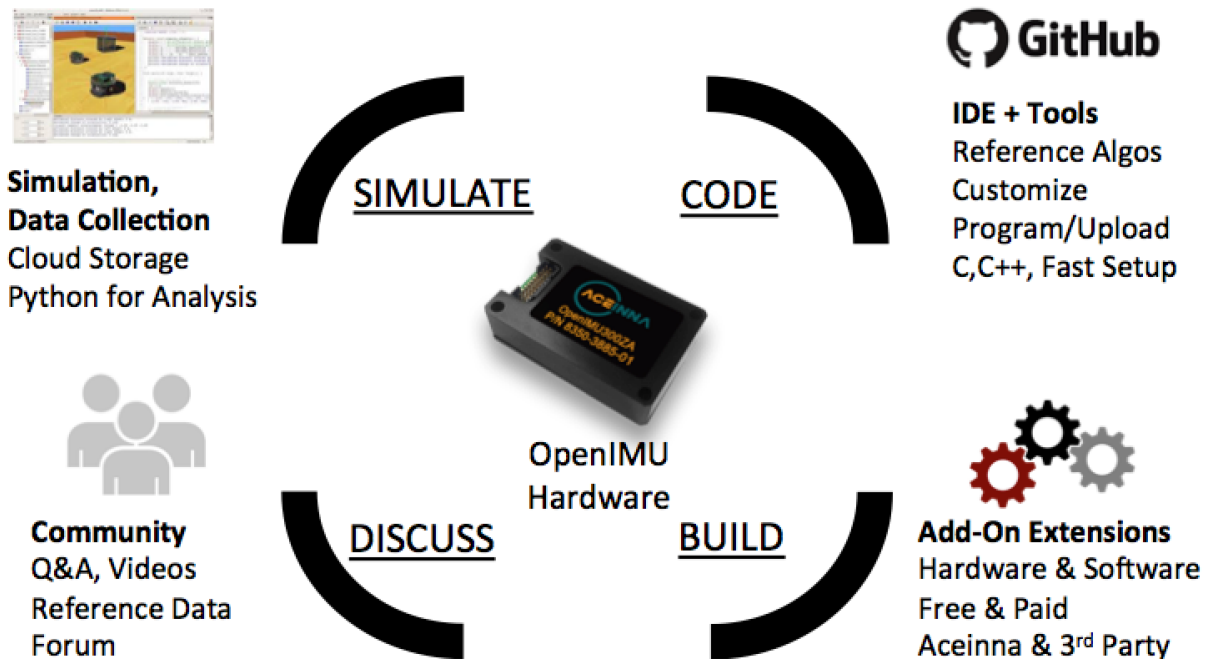
On local Ubuntu machine, you will clone the aforementioned repository and make the project. This requires the following packages to be installed:

- CMake > v2.8.7
- Gcc compiler
- Libusb v1.0

```
# Run from source directory stlink/
$make release
$cd build/Release
$sudo make install

# Plug ST-LINK/V2 into USB, and check the device is present
$ls /dev/stlink-v2
```

## 3.2 Development Tools

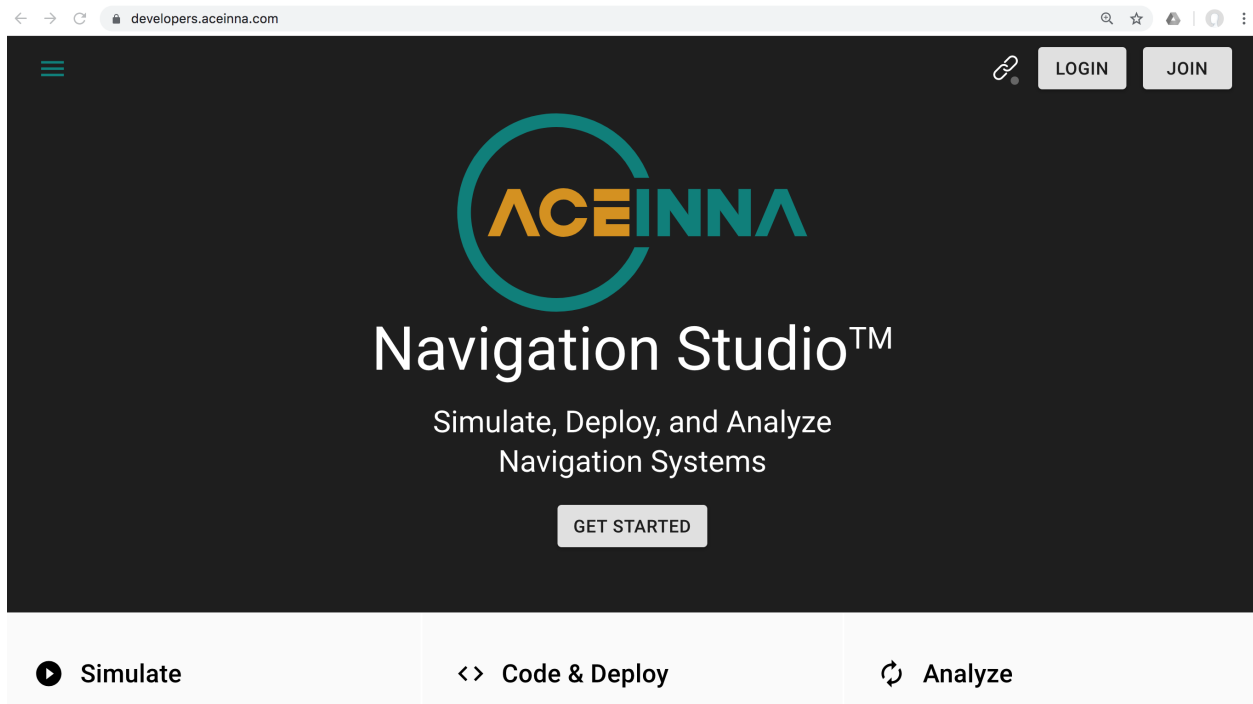


The OpenIMU development environment consists of the following main components:

- Acienna Navigation Studio (ANS)
- Visual Studio Code IDE (VSCode)

- Debugging using the PlatformIO Debugger and the JTAG Debug Adapter
- In System Firmware Update
- Python Interface
- 'openimu.json' Configuration File

### 3.2.1 Aceinna Navigation Studio



Aceinna Navigation Studio is a web-portal and UI for your OpenIMU. To run it, first ensure the Python OpenIMU driver is installed, then start the server from the command line interface as shown below.

```
$openimu
Connected ....OpenIMU300ZI - 0.0.1      SN:1808629112
```

Supported browsers are Chrome, Opera, and Edge. Firefox also works but requires an extra step described here. <https://stackoverflow.com/questions/11768221/firefox-websocket-security-issue>

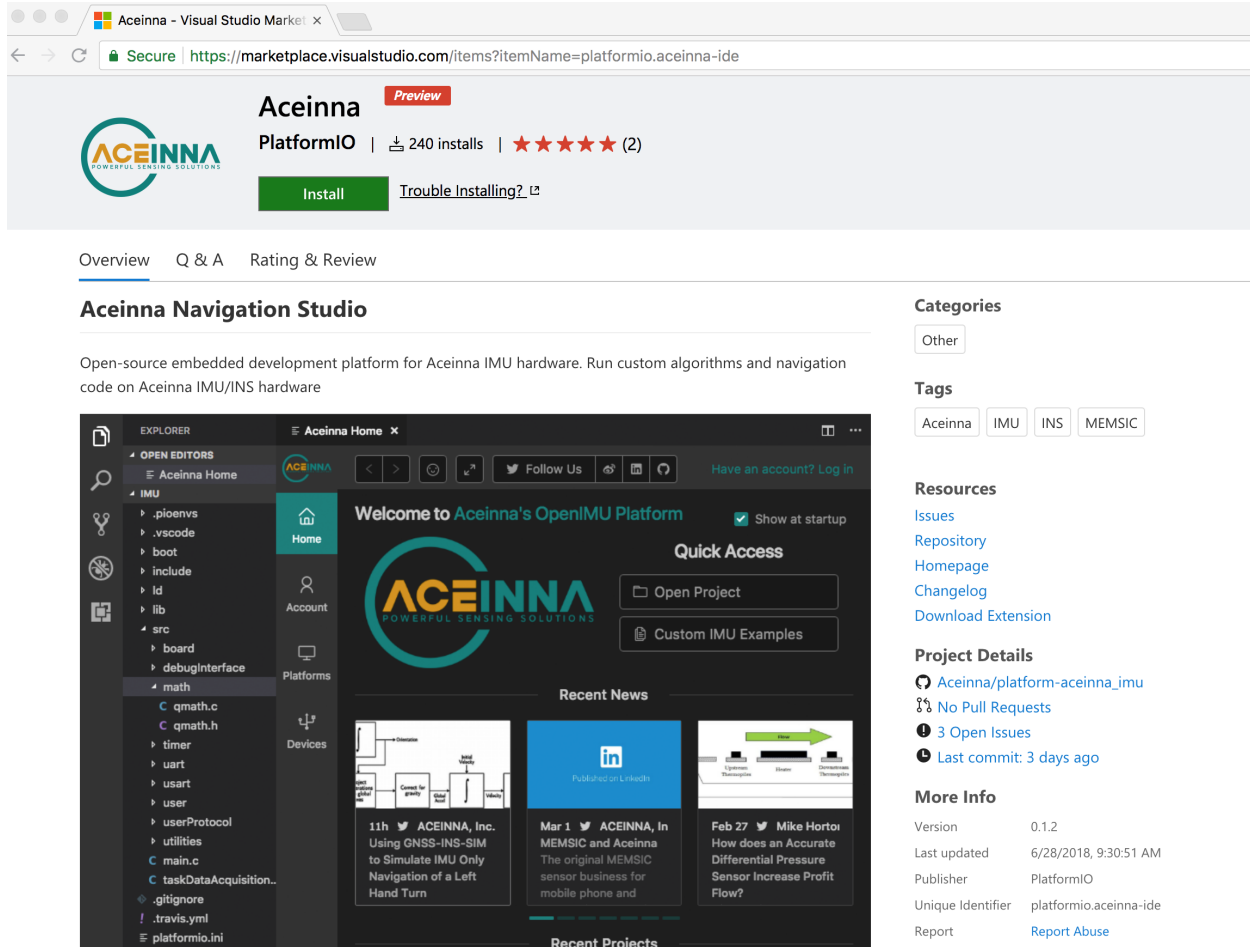
To plot data go to the link <https://developers.aceinna.com/devices/record-next> and click play. You can also log from this GUI.

The settings as well as available packet types that show up in ANS graphical user interface are controlled by *openimu.json* and their corresponding code in *userConfiguration.c*. Select the packet that you would like to display.

Once a file is logged you can retrieve the file at <https://developers.aceinna.com/devices/files>

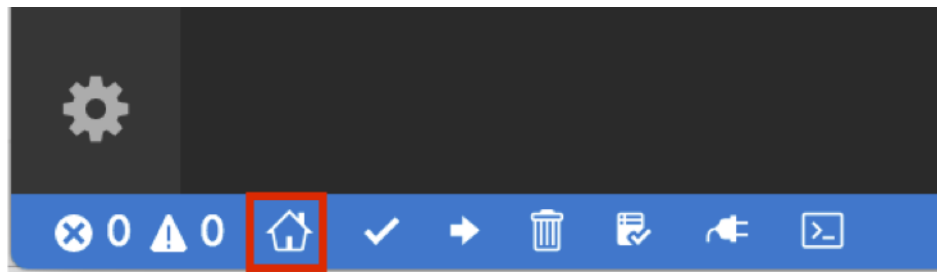
**Note:** Your data file list is only shown to you and is tied to your login credentials. The file list is not available to other users.

### 3.2.2 Visual Studio Code IDE



At the heart of the OpenIMU IDE is a custom extension built for Visual Studio Code. The installation of this extension is detailed in Quick Start. Aceinna's OpenIMU extension is a custom version of the popular open-source embedded development extension PlatformIO. PlatformIO provides many additional features including an extensive set of command line tools which are not documented on this site. Please visit <https://docs.platformio.org> for more details.

The Aceinna Visual Studio extension adds an easy to find home button at the bottom of the Visual Studio tool bar. This is shown below. Click the home button any time to return to the launch screen for embedded OpenIMU development within Visual Studio Code.



The Aceinna Visual Studio extension also automatically installs additional supporting tools. Importantly if your local system does not already have Python, the extension will install Python which enables a large number of features on the platform including serial drivers and a small server which can connect your IMU to the Aceinna Navigation Studio



developer's site for charting, graphing, and configuration.

The basic functions such as compile, clean, and upload code to device are also easily accessed from the tool bar at the bottom of the VSCode extension.

**Note:** Do not install the PlatformIO extension. Instead install the Aceinna extension. This will install all the PlatformIO tools automatically, as well as the IMU source code and Python drivers.

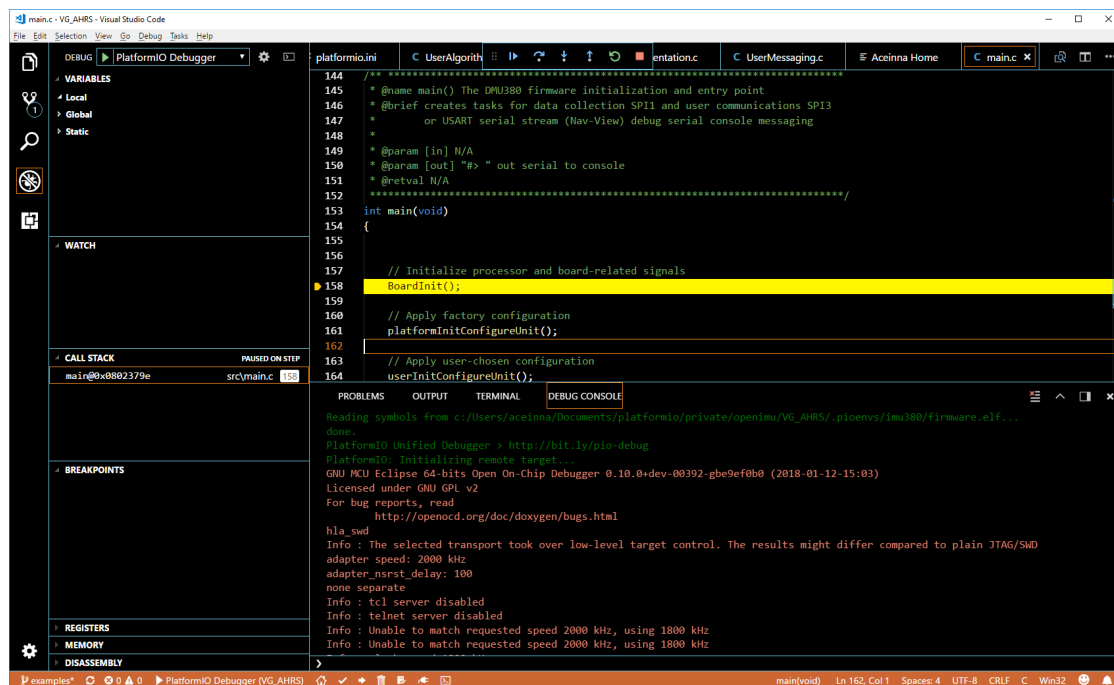
### 3.2.3 Debugging using the PlatformIO Debugger and the JTAG Debug Adapter

There are two primary methods to debug a program on OpenIMU.

- Use Visual Studio Code with ST-Link JTAG pod.
- Use the debug serial port to output debug messages.

#### 1. Debugging Using Visual Studio Code and JTAG Debugger

Visual Studio Code with installed Aceinna extension supports in-system debugging via ST-LINK JTAG pod. It allows to load and run application, stop in any place of the code by using breakpoints, observe and set values of local and global variables, observe device memory contents. The following screen shots show Visual Studio Code screen in debug mode.



Debug mode can be entered by clicking on “Debug” icon - fourth from top on very left of the screen and then clicking on green arrow “PlatformIO debugger” on top of the screen or alternatively from the menu “Debug->Start Debugging”. After entering debug mode use debug control icons on top of the screen or commands from “Debug” menu. After clicking “Debug” icon on the left of the screen while in debug mode allows to observe variables, memory, registers, call stack, etc.

#### 2. Debugging Using Debug Serial Port

User defined ASCII messages can be sent out via debug serial connection. Default baud rate is 38.4 Kbaud. One can easily change debug port baud rate in main.c file:

```
// Initialize the DEBUG USART (serial) port
InitDebugSerialCommunication(38400); // debug_usart.c
```

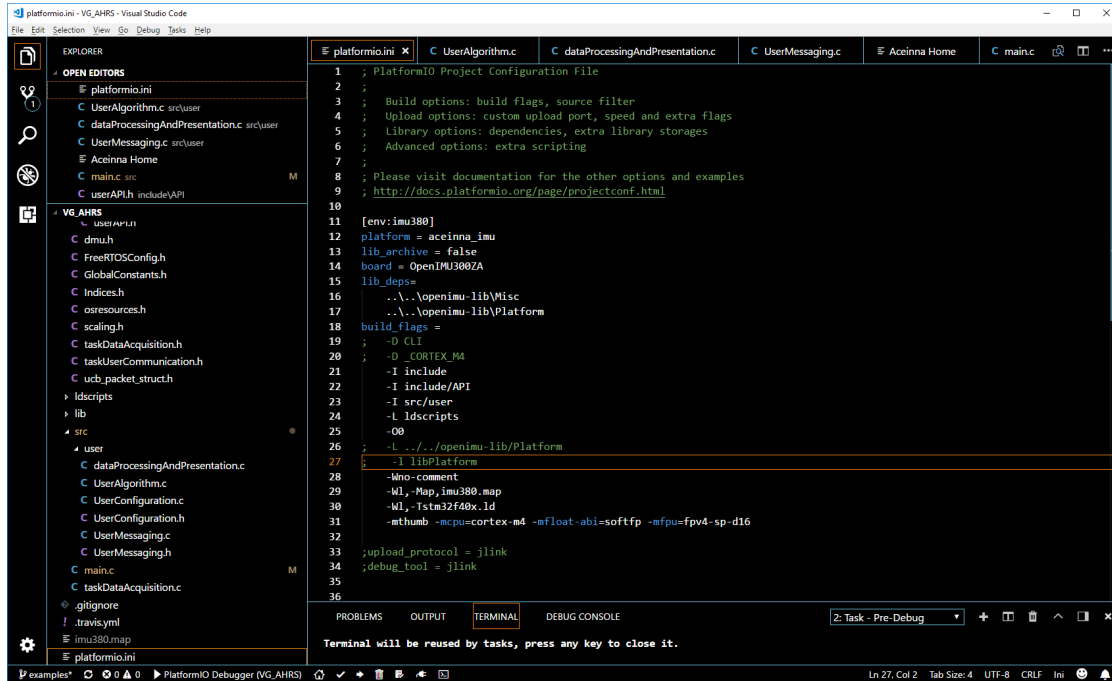
Custom printf-like syntax outputs ASCII data on debug serial port

```
int tprintf(char *format, ...);
```

Alternative macros for outputting type-specific values defined in the debug.h file.

OpenIMU unit has built-in CLI which can be enabled by uncommenting next line in file platformio.ini :

-D CLI



It allows to send custom ASCII commands to OpenIMU unit via debug serial port using any serial terminal program. CLI engine reside in CLI directory in libraries source tree. Please note that while unit connected to PC via USB port it is visible as four consecutive virtual serial ports. Third port in a row will be debug serial port.

#### Note:

- The Acienna VSCode extension uses the underlying PlatformIO debugging feature.
- PlatformIO now provides free JTAG debugging for all users.

**Note:** Visual Studio Code with installed Aceinna extension provides download of application image into device memory via JTAG by clicking “Right Arrow” icon on the bottom of the screen. This is the fastest method to download code and generally requires just a few seconds.

**Note:** The documentation and tutorials on this site assume use of the ST-LINK JTAG pod. The JTAG pod is shipped with every OpenIMU developer’s kit.

### 3.2.4 In-System Update

All OpenIMU hardware modules come shipped pre-configured with a special bootloader resident in their FLASH memory. This bootloader allows for in-system code updates using a UART connection without using JTAG. Sample code that utilizes this Bootloader can be found in the OpenIMU Python driver. An example of how to invoke the Python driver for code loading is here.

The full details of the bootloader serial protocol is described below. These commands are executed using OpenIMU's standard serial interface:

#### Bootloader Initialization

A user can initiate bootloader at any time by sending 'JI' command (see below command's format) to application program. This command forces the unit to enter bootloader mode. The unit will communicate at 57.6Kbps baud rate regardless of the original baud rate the unit is configured to. The Bootloader always communicates at 57.6Kbps until the firmware upgrade is complete.

As an additional device recovery option immediately after powering up, every OpenIMU will enter a recovery window of 100ms prior to application start. During this 100mS window, the user can send 'JI' command at 57.6Kbs to the Bootloader in order to force the unit to remain in Bootloader mode.

Once the device enters Bootloader mode via the 'JI' command either during recovery window or from normal operation, a user can send a sequence of 'WA' commands to write a complete application image into the device's FLASH.

After loading the entire firmware image with successive 'WA' commands, a 'JA' command is sent to instruct the unit to exit Bootloader mode and begin application execution. At this point the device will return to its original baud rate.

Optionally, the system can be rebooted by toggling the power or toggling nRst (pull low and release) to restart the system.

#### Firmware Update Commands

The commands detailed below are used for upgrading a new firmware version via the UART at 57.6Kbps.

##### Jump to Bootloader Command

Jump To Bootloader ('JI'=0x4A49)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x4A49	0x00		CRC(U2)

The command allows system to enter bootloader mode.

##### Write App Command

Write APP ('WA'=0x5741)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x5741	len+5		CRC(U2)

The command allows users to write binary sequentially to flash memory in bootloader mode. The total length is the sum of payload's length and 4-byte address followed by 1-byte data length. See the following table of the payload's format.

WA Payload Contents					
Byte Offset	Name	Format	Scaling	Units	Description
0	staringAddr	U4	•	bytes	The FLASH word offset to begin writing data
4	byteLength	U1	•	bytes	The word length of the the data to write
5	dataByte0	U1	•	•	Flash data
6	dataByte1	U1	•	•	Falsh data
...	...				
4+byteLength	dataByte	U1	•	•	Flash data

Payload starts from 4-byte address of flash memory where the binary is located. The fifth byte is the number of bytes of dataBytess, but less than 240 bytes. User must truncate the binary to less than 240-byte blocks and fill each of blocks into payload starting from the sixth-byte. See the reference code, function `write_block()`, in Appendix F.

### Jump to Application Command

Jump To Application ('JA'=0x4A41)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x4A41	0x00		CRC(U2)

The command allows system directly to enter application mode.

## 3.2.5 Python Interface

The OpenIMU Python driver supports communication with the hardware for data logging and device configuration over the main user UART interface of the OpenIMU hardware. When run in server mode, it allows connection of the OpenIMU with the developer's website Aceinna Navigation Studio and its friendly GUI interface.

The Python driver attempts to automatically find a connected OpenIMU hardware by scanning available ports at various baud rates. Once a connection is established, this connection is recorded in a file named *connection.json*. On the next use of the driver, the driver will first attempt communication on this port speeding up connection time.

The Python driver reads a JSON file by default named *openimu.json* to understand the messages - both primary output packets, as well as command/response type packets from the IMU. These can be customized by changing the JSON file and the Python script will use that information to parse data (literally the byte stream) from the OpenIMU in real-time appropriately.

Here are a few samples function you can call with the driver.

```

# Create a device and connect to it
imu = OpenIMU()
imu.find_device()

# Get all parameters by issuing 'gA' command
imu.openimu_get_all_param()

# Update a parameter by issuing 'uP' command
# See openimu.json for the parameter numbers
# This example changes output packet rate to 100Hz
imu.openimu_update_param(4,100)

# Save parameter changes by issuing 'sC' command
imu.openimu_save_config()

# Log data for 1Hr
# Data is logged into data directory with time of day string as default filename
imu.start_log()
time.sleep(3600)
imu.stop_log()

# Update units firmware
# bin file is stored in .pioenvs directory and created after compilation
# the file must be moved to where the Python driver can find it
imu.openimu_upgrade_fw('myapp.bin')

```

You can also run the python code as a CLI interface to the unit. The CLI is defined in `commands.py`. If you have installed the python driver with `pip install`, then navigate to a directory that contains a valid `openimu.json` for your unit, and you can type:

```

$openimu
Connected ....OpenIMU300ZI - 0.0.1      SN:1808629112
>>help
Usage:
help : CLI help menu
exit : exit CLI
run : Operations defined by users
save : Save the configuration into EEPROM
connect : Find OpenIMU device
upgrade : Upgrade firmware
record : Record output data of OpenIMU on local machine
stop : stop recording outputs on local machine
server_start : start server thread and must use exit command to quit
get : Read the current configuration and output data
set : Write parameters to OpenIMU
>>

```

**Note:** As you develop code and customize your OpenIMU, you should also update `openimu.json` to keep it in sync with your changes. This way both the Python driver and developers website, ANS, will function properly and understand your units special programmed characteristics. The `openimu.json` file updates the Python driver functions as well as the ANS website UI.

### 3.2.6 openimu.json Configuration File

The *openimu.json* file is used to describe the input and output messages and the configuration parameters of the OpenIMU. An example file is shown below. The two sections that are edited during development are “userConfiguration” and “userMessages”. These sections of the JSON file correspond to equivalent sections of code in the your custom application. The description provided in the *openimu.json* file is used by the Python driver to support additional configuration parameters and messages that you add to your unit. For example, if you add a custom output message, the Python driver can automatically log it in a properly delimited CSV file format. In addition, the *openimu.json* file provides user friendly names and features that then appear in the ANS website automatically. Using the same custom output message as an example, the *openimu.json* file can describe the graphs and plots that are shown on the “Record” page of the website. The *openimu.json* file lets you reuse driver and UI code with little or no modification.

In the main OpenIMU source tree, you will find the “user” directory for your project. This is where your custom IMU app code is integrated and built. The files *userConfiguration.h/userConfiguration.c* describes the various configuration parameters in the unit. The files *userMessaging.h/userMessaging.c* describes both the default and custom messages for your OpenIMU app. These sections of c-code are then described in the “userConfiguration” and “userMessages” section of in *openimu.json* as shown below. If you add a new parameter in *userConfiguration.c*, then you add a new parameter in “userConfiguration” following the examples. Note each parameter must have a unique “paramId”. If you add a unique output message, you will add that both to the “Packet Type” options array, and as a new “outputPacket” in “userMessages”. When adding a new message the key point is to properly describe the payload in the order that the data is sent in *userMessaging.c*.

```
{
  "name" : "OpenIMU300-EZ",
  "type" : "openimu",
  "description" : "9-axis OpenIMU with triaxial rate, acceleration, and magnetic_
↪ measurement",
  "userConfiguration" : [
    { "paramId": 0, "paramType" : "disabled", "type" : "uint64", "name": "Data CRC" ↪
↪ },
    { "paramId": 1, "paramType" : "disabled", "type" : "uint64", "name": "Data Size
↪ " },
    { "paramId": 2, "paramType" : "select", "type" : "int64", "name": "Baud Rate",
↪ "options" : [38400, 57600, 115200]},
    { "paramId": 3, "paramType" : "select", "type" : "char8", "name": "Packet Type",
↪ "options" : ["z1", "zT"]},
    { "paramId": 4, "paramType" : "select", "type" : "int64", "name": "Packet Rate",
↪ "options" : [200, 100, 50, 20, 10, 0]},
    { "paramId": 5, "paramType" : "select", "type" : "int64", "name": "Accel LPF",
↪ "options" : [50, 25, 40, 20, 10, 5, 2]},
    { "paramId": 6, "paramType" : "select", "type" : "int64", "name": "Rate LPF",
↪ "options" : [50, 25, 40, 20, 10, 5, 2]},
    { "paramId": 7, "paramType" : "select", "type" : "char8", "name": "Orientation",
↪ "options" : ["+X+Y+Z"]}
  ],
  "userMessages" : {
    "inputPackets" : [
      {
        "name" : "pG",
        "description" : "Get device serial number & factory ID",
        "inputPayload" : {
        },
        "responsePayload" : {
          "type" : "string",
          "name" : "Device ID and SN"
        }
      }
    ]
  }
}
```

(continues on next page)

(continued from previous page)

```

    {
      "name" : "gV",
      "description" : "Get user app version",
      "inputPayload" : {},
      "responsePayload" : {
        "type" : "string",
        "name" : "User Version"
      }
    },
    {
      "name" : "gA",
      "description" : "Get All Configuration Parameters",
      "inputPayload" : {},
      "responsePayload" : {
        "type" : "userConfiguration",
        "name" : "Full Current Configuration"
      }
    },
    {
      "name" : "gP",
      "description" : "Get a Configuration Parameter",
      "inputPayload" : {
        "type" : "paramId",
        "name" : "Request Parameter Id"
      },
      "responsePayload" : {
        "type" : "userParameter",
        "name" : "User Parameter"
      }
    },
    {
      "name" : "sC",
      "description" : "Save Configuration Parameters to Flash",
      "inputPayload" : {},
      "responsePayload" : {}
    },
    {
      "name" : "uP",
      "description" : "Update Configuration Parameter",
      "inputPayload" : {
        "type" : "userParameter",
        "name" : "Parameter to be Updated"
      },
      "responsePayload" : {
        "type" : "paramId",
        "name" : "ID of the Updated Parameter"
      }
    }
  ],
  "outputPackets" : [
    {
      "name": "z1",
      "description": "Scaled 9-Axis IMU",
      "payload" : [
        {
          "type" : "uint32",
          "name" : "time",

```

(continues on next page)

(continued from previous page)

```

        "unit" : "s"
    },
    {
        "type" : "float",
        "name" : "xAccel",
        "unit" : "G"
    },
    {
        "type" : "float",
        "name" : "yAccel",
        "unit" : "G"
    },
    {
        "type" : "float",
        "name" : "zAccel",
        "unit" : "G"
    },
    {
        "type" : "float",
        "name" : "xRate",
        "unit" : "deg/s"
    },
    {
        "type" : "float",
        "name" : "yRate",
        "unit" : "deg/s"
    },
    {
        "type" : "float",
        "name" : "zRate",
        "unit" : "deg/s"
    },
    {
        "type" : "float",
        "name" : "xMag",
        "unit" : "Gauss"
    },
    {
        "type" : "float",
        "name" : "yMag",
        "unit" : "Gauss"
    },
    {
        "type" : "float",
        "name" : "zMag",
        "unit" : "Gauss"
    }
},
"graphs" : [
    {
        "name" : "Acceleration",
        "units" : "m/s/s",
        "xAxis" : "Time (s)",
        "yAxes" : [ "xAccel", "yAccel", "zAccel"],
        "colors" : [ "#FF0000", "#00FF00", "#0000FF" ],
        "yMax" : 80
    }
],

```

(continues on next page)



(continued from previous page)

```

        {
            "name" : "Angular Rate",
            "units" : "deg/s",
            "xAxis" : "Time (s)",
            "yAxes" : [ "xRate", "yRate", "zRate"],
            "colors" : [ "#FF0000", "#00FF00", "#0000FF" ],
            "yMax" : 400
        }
    ]
},
{
    "name": "z2",
    "description": "Arbitrary type Values",
    "payload" : [
        {
            "type" : "uint32",
            "name" : "time",
            "unit" : "s"
        },
        {
            "type" : "uchar",
            "name" : "c",
            "unit" : ""
        },
        {
            "type" : "int16",
            "name" : "s",
            "unit" : ""
        },
        {
            "type" : "int32",
            "name" : "i",
            "unit" : ""
        },
        {
            "type" : "int64",
            "name" : "ll",
            "unit" : ""
        },
        {
            "type" : "double",
            "name" : "d",
            "unit" : ""
        }
    ],
    "graphs" : [
        {
            "name" : "Angular Rate",
            "units" : "deg/s",
            "xAxis" : "Time (s)",
            "yAxes" : [ "xRate", "yRate", "zRate"],
            "colors" : [ "#FF0000", "#00FF00", "#0000FF" ],
            "yMax" : 400
        }
    ]
},
{

```

(continues on next page)

(continued from previous page)

```

    "name": "z3",
    "description": "Scaled 6-Axis IMU Values",
    "payload" : [
      {
        "type" : "int",
        "name" : "timestamp",
        "unit" : "ms"
      },
      {
        "type" : "float",
        "name" : "xAccel",
        "unit" : "m/s/s"
      },
      {
        "type" : "float",
        "name" : "yAccel",
        "unit" : "m/s/s"
      },
      {
        "type" : "float",
        "name" : "zAccel",
        "unit" : "m/s/s"
      },
      {
        "type" : "float",
        "name" : "xRate",
        "unit" : "rad/s"
      },
      {
        "type" : "float",
        "name" : "yRate",
        "unit" : "rad/s"
      },
      {
        "type" : "float",
        "name" : "zRate",
        "unit" : "rad/s"
      }
    ],
    "graphs" : [
      {
        "name" : "Acceleration",
        "units" : "m/s/s",
        "xAxis" : "timestamp (ms)",
        "yAxes" : [ "xRate", "yRate", "zRate" ],
        "colors" : [ "#FF0000", "#00FF00", "#0000FF" ],
        "yMax" : 100
      }
    ]
  },
  "bootloaderMessages": [
    {
      "name" : "JI",
      "description" : "Jump to Bootloader",
      "inputPayload" : {},

```

(continues on next page)

(continued from previous page)

```

        "responsePayload" : {
            "type" : "ack",
            "response" : "Acknowledgement"
        }
    },
    {
        "name" : "JA",
        "description" : "Jump to App",
        "inputPayload" : {},
        "responsePayload" : {
            "type" : "none",
            "response" : "Empty"
        }
    },
    {
        "name" : "WA",
        "description" : "Write App Block",
        "inputPayload" : {
            "type" : "block",
            "name" : "4 byte block address followed by up to 240 bytes data"
        },
        "responsePayload" : {
            "type": "ack",
            "response" : "Acknowledgement"
        }
    }
}
]
}

```

---

**Note:** Don't modify the "bootloaderMessages" section of *openimu.json*. This section is used by the Python driver for the in-system programming bootloader. It should not be changed

---

The easy way to get started quickly is to purchase an OpenIMU Developer's Kit from Aceinna <https://www.aceinna.com> or a local distributor. The developer's kit includes an OpenIMU300EZ inertial measurement unit, JTAG Pod, Eval board, and precision test fixture. The precision test fixture makes it easy to properly align and install the IMU in a target vehicle for integration testing.

---

## Ready-to-Use Applications

---

OpenIMU ships with a number of ready to use, downloadable applications to help the user get started. These apps can be compiled without modification and downloaded to your unit. All OpenIMU modules by default ship with the IMU app described on the IMU App page.

To learn about ready to use apps available for immediate download to your OpenIMU, please see the the following page: [Aceinna Navigation Studio - Getting Started](#)

---

**Note:** Use the browser back button to return to the OpenIMU documentation.

---

To install ready-made apps to your IMU, please make sure the user have installed the OpenIMU python driver described in the “Development Tools - Python Interface” subsection and started the server.

To build a custom app, please follow the tutorial provided later in the OpenIMU documentation at “Tutorial - What The User Needs to Know to Build The First Application”

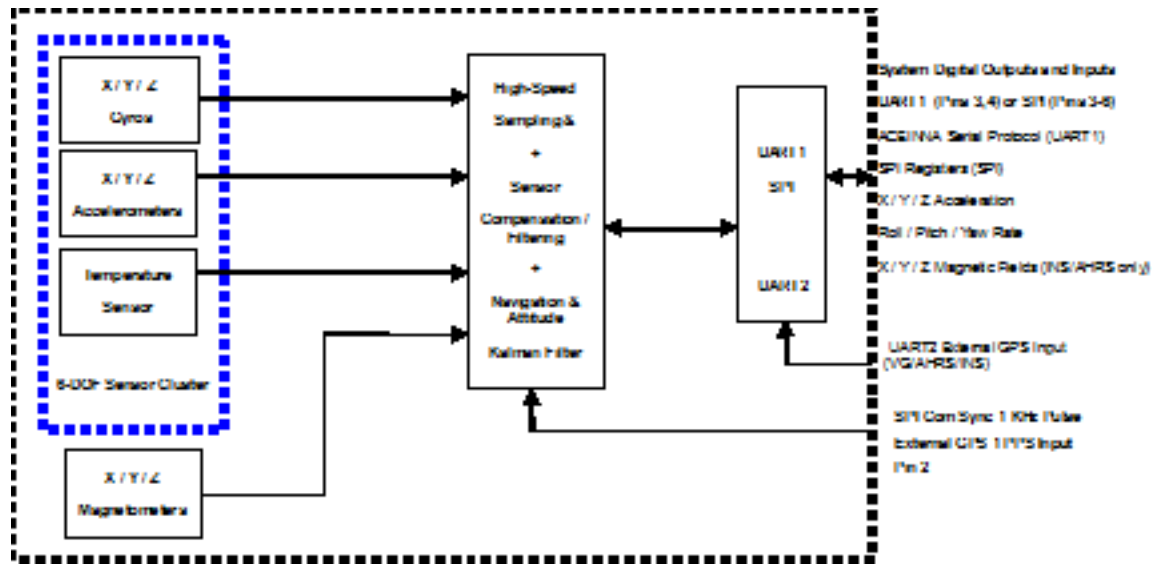
The following Ready-To-Use Applications are available:

- Inertial Measurement Unit (IMU) App
- Leveler App
- AHRS/VG Dynamic Attitude App
- GPS/INS App

### 4.1 IMU App

The App name, IMU, stands for Inertial Measurement Unit, and the name is indicative of the basic inertial measurement unit functionality provided by this APP. The IMU App signal processing chain consists of high-speed sampling of the 9-DOF sensor cluster (accelerometers, rate sensors, and magnetometers), programmable low-pass filters, and the execution of built-in calibration models.

Additionally any configuration parameters settings such as axes rotation are applied to the IMU data. The 200Hz IMU data is continuously being maintained inside the IMU APP, and is Digital IMU data is output over the UART port at a selectable fixed rate (200, 100, 50, 25, 20, 10, 5 or 2 Hz). The digital IMU data is available in one of several measurement packet formats including Scaled Sensor Data ('z1' Packet).



## 4.2 Leveler App

Leveler App Description - To Be Provided

## 4.3 AHRS/VG Dynamic Attitude App

The Attitude and Heading Reference System (AHRS) and Vertical Gyro (VG) application supports all of the features and operating modes of the IMU APP, and it links in additional internal software, running on the processor, for the computation of dynamic roll, pitch. In addition to the Roll, Pitch and IMU data, the dynamic heading measurement is optionally stabilized using the 3-axis magnetometer as a magnetic north reference. Roll, Pitch measurements are often referred to as “VG” or Vertical Gyro measurements. When heading stabilized by a magnetometer is added, the solution is often referred to as an “AHRS” or Attitude Heading Reference System. Hence the name of this APP is AHRS/VG APP.

At a fixed 200Hz rate, the VG/AHRS APP continuously maintains the digital IMU data as well as the dynamic roll, pitch, and heading. As shown in diagram after the Sensor Calibration Block, the IMU data is passed to the Integration to Orientation block. The Integration to Orientation block integrates body frame sensed angular rate to orientation at a fixed 200 times per second within all of the OpenIMU Series products.

As also shown in the software block diagram, the Integration to Orientation block receives drift corrections from the Extended Kalman Filter or Drift Correction Module. In general, rate sensors and accelerometers suffer from bias drift, misalignment errors, acceleration errors (g-sensitivity), nonlinearity (square terms), and scale factor errors. The largest error in the orientation propagation is associated with the rate sensor bias terms. The Extended Kalman Filter (EKF) module provides an on-the-fly calibration for drift errors, including the rate sensor bias, by providing corrections to the Integration to Orientation block and a characterization of the gyro bias state. In the AHRS/VG APP, the internally computed gravity reference vector and the distortion corrected magnetic field vector provide an attitude reference

measurement for the EKF when the unit is in quasi-static motion to correct roll, pitch, and heading angle drift and to estimate the X, Y and Z gyro rate bias. The AHRS/VG APP adaptively tunes the EKF feedback gains in order to best balance the bias estimation and attitude correction with distortion free performance during dynamics when the object is accelerating either linearly (speed changes) or centripetally (false gravity forces from turns). Because centripetal and other dynamic accelerations are often associated with yaw rate, the AHRS/VG APP maintains a low-passed filtered yaw rate signal and compares it to the turnSwitch threshold field (user adjustable). When the user platform exceeds the turnSwitch threshold yaw rate, the AHRS/VG APP lowers the feedback gains from the accelerometers to allow the attitude estimate to coast through the dynamic situation with primary reliance on angular rate sensors. This situation is indicated by the softwareStatus - turnSwitch status flag. Using the turn switch maintains better attitude accuracy during short-term dynamic situations, but care must be taken to ensure that the duty cycle of the turn switch generally stays below 10% during the vehicle mission. A high turn switch duty cycle does not allow the system to apply enough rate sensor bias correction and could allow the attitude estimate to become unstable.

The AHRS/VG APP algorithm also has two major phases of operation. The first phase of operation is the attitude initialization phase. During the initialization phase, the OpenIMU unit is expected to be stationary or quasi-static to rapidly estimate the X, Y, and Z rate sensor bias, and the initial attitude. The initialization phase lasts approximately 2 seconds. After the initialization phase, the EKF algorithm in the AHRS/VP APP dynamically tunes the feedback (also referred to as EKF gain) from the accelerometers and magnetometers to continuously estimate and correct for roll, pitch, and heading (yaw) errors, as well as to estimate X, Y, and Z rate sensor bias.

### 4.3.1 The Definitions of The Output Packets of The VG/AHRS App

#### “a1” packet

The default VG/AHRS app output packet type is “a1”, and it is defined in the following two tables.

(‘a1’ = 0x6131)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6131	47		<CRC (U2)>

Payload:

Byte Offset	Name	Format	Notes
0	System Timer of sensors sampling	U4	LSB First msec
4	Above timer converted to a double type	D	LSB First second
12	Roll	F4	LSB First deg
16	Pitch	F4	LSB First deg
20	corrected X gyro	F4	LSB First deg/s
24	corrected Y gyro	F4	LSB First deg/s
28	corrected Z gyro	F4	LSB First deg/s
32	X Accel	F4	LSB First m/s/s
36	Y Accel	F4	LSB First m/s/s
40	Z Accel	F4	LSB First m/s/s
44	Operation mode <sup>1</sup>	U1	LSB First
45	Linear accel switch <sup>2</sup>	U1	LSB First
46	Turn switch <sup>3</sup>	U1	LSB First

<sup>1</sup> Operation mode of the algorithm. 0 for waiting for the system to stabilize, 1 for initializing attitude, 2 and 3 for VG/AHRS mode, and 4 for INS

**“a2” packet**

If you want to output the yaw angle, you can choose the “a2” packet. For the VG app, the yaw angle is from integrating the gyro rate, and for the AHRS app, the yaw angle gets corrected by magnetometer measurements.

('a2' = 0x6132)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6132	48		<CRC (U2)>

Payload:

---

mode. Please refer to the source code for details.

<sup>2</sup> 0 if linear acceleration is detected, 1 if no linear acceleration. Please refer to the source code for details.

<sup>3</sup> Indicate if the filtered yaw rate exceeds the turn switch threshold. 1 yes, 0 no. Please refer to the source code for details.



Byte Offset	Name	Format	Notes
0	System Timer of sensors sampling	U4	LSB First msec
4	Above timer converted to a double type	D	LSB First second
12	Roll	F4	LSB First deg
16	Pitch	F4	LSB First deg
20	Yaw	F4	LSB First deg
24	corrected X gyro	F4	LSB First deg/s
28	corrected Y gyro	F4	LSB First deg/s
32	corrected Z gyro	F4	LSB First deg/s
36	X Accel	F4	LSB First m/s/s
40	Y Accel	F4	LSB First m/s/s
44	Z Accel	F4	LSB First m/s/s

**“e1” packet**

If you further want to output the magnetometer measurements, you can choose the “e1” packet.

('e1' = 0x6531)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6531	75		<CRC (U2)>

Payload:

Byte Offset	Name	Format	Notes
0	System Timer of sensors sampling	U4	LSB First msec
4	Above timer converted to a double type	D	LSB First second
12	Roll	F4	LSB First deg
16	Pitch	F4	LSB First deg
20	Yaw	F4	LSB First deg
24	X Accel	F4	LSB First g
28	Y Accel	F4	LSB First g
32	Z Accel	F4	LSB First g
36	X gyro	F4	LSB First deg/s
40	Y gyro	F4	LSB First deg/s
44	Z gyro	F4	LSB First deg/s
48	X gyro bias	F4	
<b>4.3. AHRS/VG Dynamic Attitude App</b>			LSB First deg/s
52	Y gyro bias	F4	

---

**Note:** In AHRS mode for proper operation of the stabilized heading measurement, the AHRS/VG APP uses information from the internal 3-axis digital magnetometer. The AHRS APP must be installed correctly and calibrated for hard-iron and soft iron effects to avoid any system performance degradation.

---

## 4.4 GPS/INS App

The INS APP supports all of the features and operating modes of the VG/AHRS APP, and it includes additional capability of interfacing with an external GPS receiver and associated software running on the processor, for the computation of navigation information as well as orientation information. The APP name, GPS/INS APP, stands for Inertial Navigation System, and it is indicative of the navigation reference functionality that APP provides by outputting inertially-aided navigation information (Latitude, Longitude, and Altitude), inertially-aided 3D velocity information, as well as heading, roll, and pitch measurements, in addition to digital IMU data.

The processor performs time-triggered trajectory propagation at 100Hz and will synchronize the sensor sampling with the GPS UTC (Universal Coordinated Time) second boundary when available.

As with the AHRS/VG APP, the algorithm has two major phases of operation. Immediately after power-up, the INS APP uses the accelerometers to compute the initial roll and pitch angles. During the first 60 seconds of startup, the INS APP should remain approximately motionless in order to properly initialize the rate sensor bias. The initialization phase lasts approximately 60 seconds, and the initialization phase can be monitored in the operation mode transmitted by default in each measurement packet.

After initialization phase, the OpenIMU continuously maintains the digital IMU data; the dynamic roll, pitch, and heading data; as well as the navigation data. The body frame sensed angular rate is first integrated to orientation at a fixed N times per second. For improved accuracy and to avoid singularities when dealing with the cosine rotation matrix, a quaternion formulation is used in the algorithm to provide attitude propagation. Using the attitude, the body frame accelerometer signals are rotated into the NED frame and integrated to velocity. And then, NED velocity is integrated to get position. At this point, the data is blended with GPS position and velocity data in the EKF, and output as a complete navigation solution.

The INS APP blends GPS derived heading and accelerometer measurements into the EKF update depending on the health and status of the associated sensors. If the GPS link is lost or poor, the Kalman Filter solution stops tracking accelerometer bias, but the algorithm continues to apply gyro bias correction and provides stabilized angle outputs. The EKF tracking states are reduced to angles and gyro bias only. The accelerometers will continue to integrate velocity, however, accelerometer noise, bias, and attitude error will cause the velocity estimates to start drifting within a few seconds. The attitude tracking performance will degrade, the heading will freely drift, and the filter will revert to the VG only EKF formulation. The UTC packet synchronization will drift due to internal clock drift.

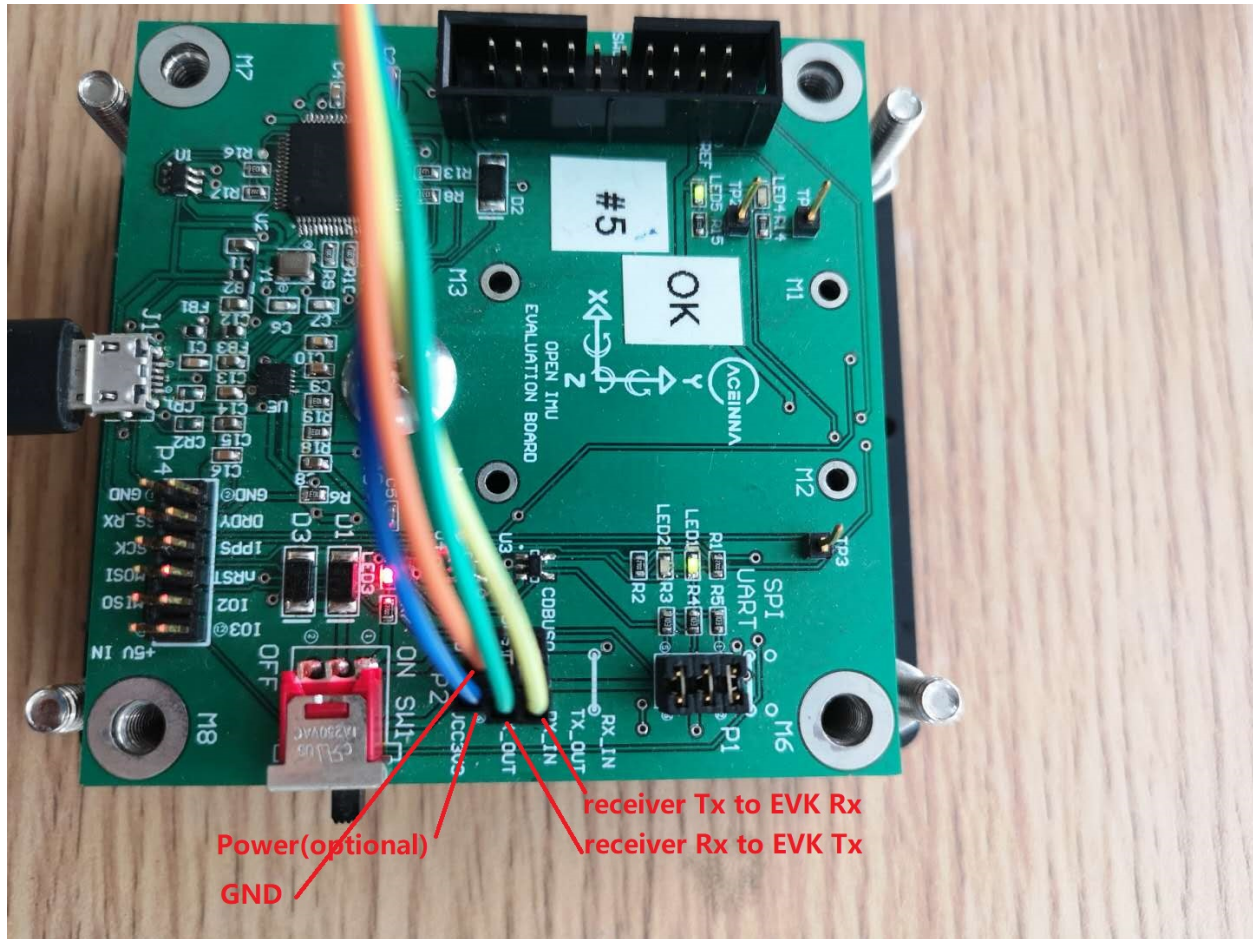
### 4.4.1 Quick Start

In this section, we explain how to get the INS app running with an external GPS receiver that outputs NMEA GGA, VTG and RMC messages. The default baud rate for UART is 115200. Although NMEA is not recommended in our INS app due to lack of some required information of the algorithm, it is chosen here because its popularity and simplicity. Our GPS driver supports NMEA message decoding, so you don't need to write a single line of code.

It is assumed that you are using our *OpenIMU300ZI EVK*.

#### Connect the GPS receiver to the EVK

In the following picture, the onboard 3.3V and GND are used to power the GPS receiver. You can also choose your own power supply.



### Burn the INS App into The Unit

The unit has a built-in IMU app. The INS app need loaded by yourself. There are two recommended ways to do that.

#### Using the Python Driver

This is for people who only want to use the precompiled bin file.

The *Python Driver* loads the INS app by the built-in bootloader of the OpenIMU300ZI unit. Please follow steps below.

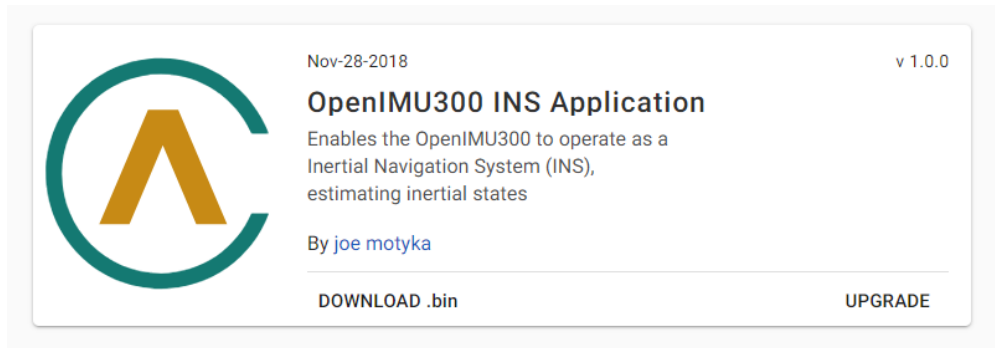
1. Connect the unit to the Python Driver.

Please refer to *Python Interface*. If the unit is successfully connected, you will see information like this.

```
Select C:\Users\liyifan\Documents\WeChat Files\silva_lee\FileStorage\File\2019-08\server_win32 1721.exe
downloading config json files from github, please waiting for a while
autoconnected
Connected ....OpenIMU300ZA 5020-3885-01 1.1.0 SN:1908400158
```

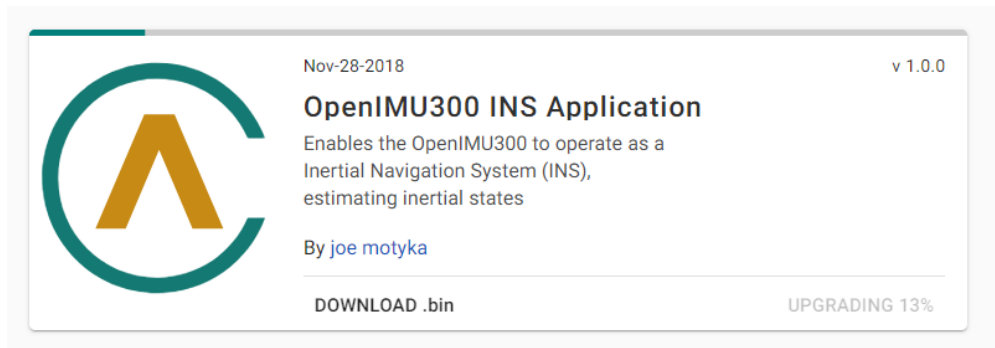
2. Vistit the App page of our Developer Site.

You can get access to all available apps in our [Developer Site](#). The OpenIMU300ZI INS app is the one you need.



### 3. Burn the INS app.

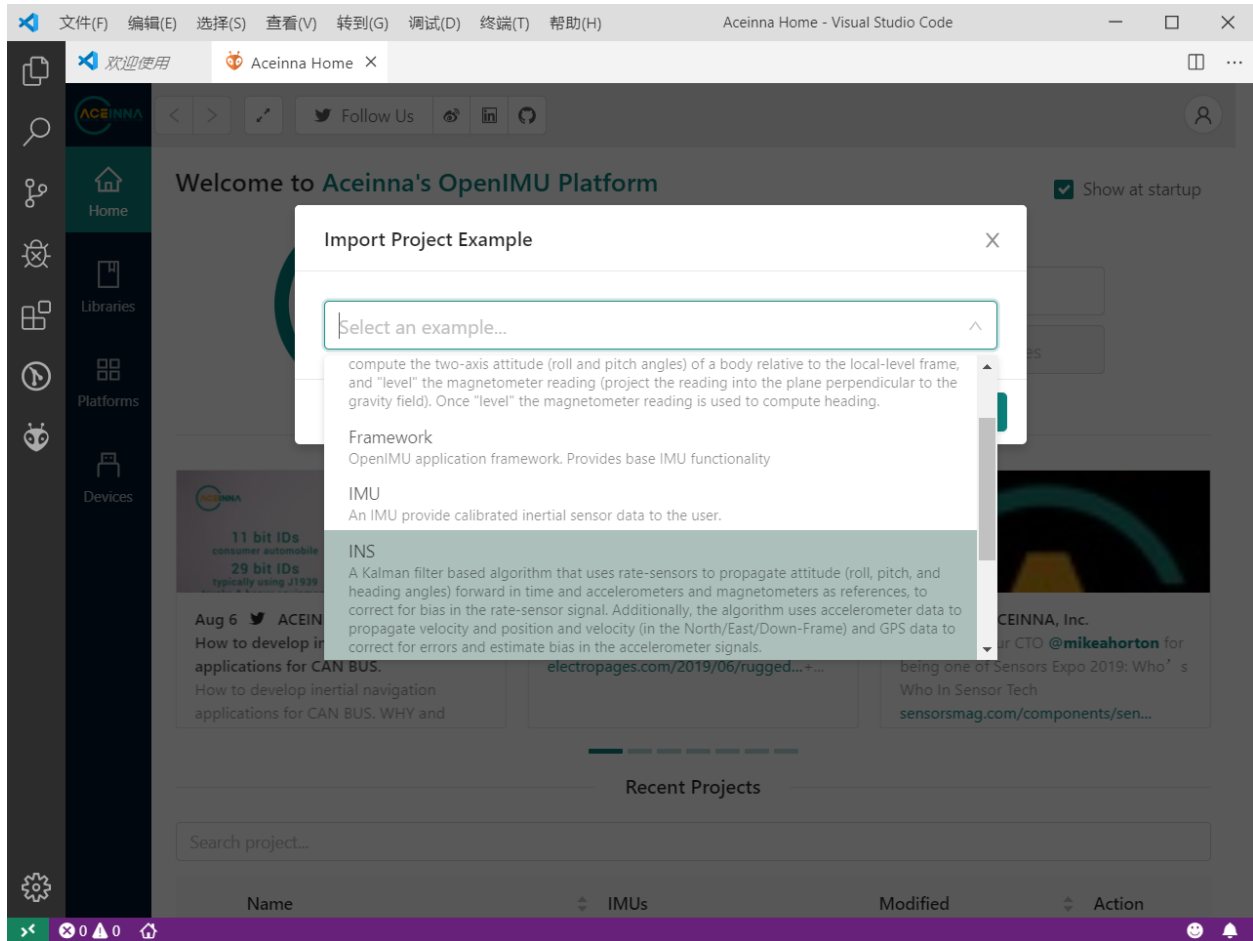
Click “UPGRADE” and wait for it to complete.



### Using Aceinna Extension in VS Code

If you want to modify our open-source code, you may want to try this way.

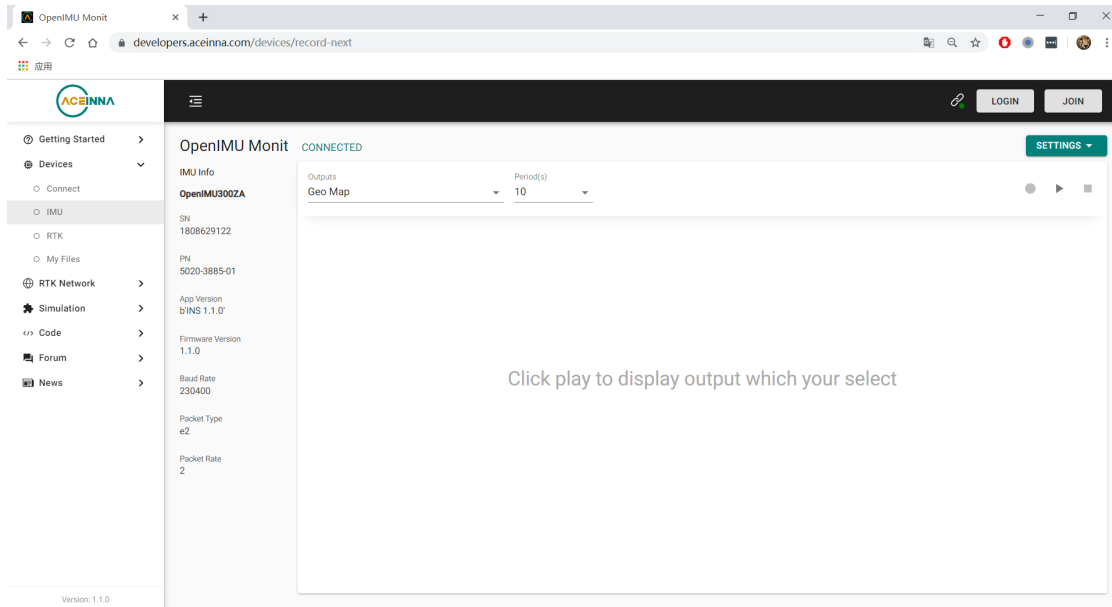
Please first refer to [PC Tools Installation](#) to install required tools and then to [Aceinna Extension](#) for basic usage of the extension. After importing the project of the INS app, you can modify the code, compile the project and upload the bin file to the unit via ST-Link.



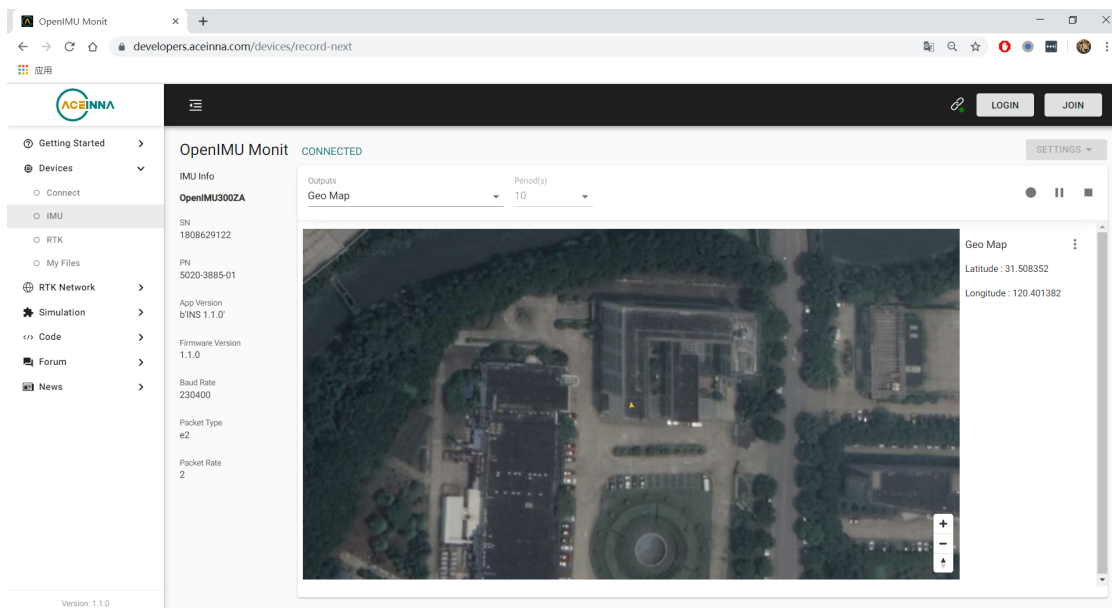
## Get and Visualize the Output

1. Connect the unit to the Python Driver.
2. Visit our [Developer Site](#).

You can see the detailed information about the unit.



Choose “Geo Map” as output, and click the play button, and you can see the live position on the map.



#### 4.4.2 How to Add Support of a New GPS Receiver Protocol

Currently we support NMEA, uBlox Nav-PVT and NovaTel Bestpos/Bestvel. If your receiver protocol is not in the list, it is easy for you to add code to decode a new protocol. Let's take uBlox nav-pvt for example to explain how to do this.

1. define the name (UBLOX\_BINARY) of the protocol in GlobalConstas.h.

```
// Choices for GPS protocol type
typedef enum{
    AUTODETECT           = -1,
    UBLOX_BINARY         = 0,
```

(continues on next page)



(continued from previous page)

```

NOVATEL_BINARY      = 1,
NOVATEL_ASCII       = 2,
NMEA_TEXT           = 3,
DEFAULT_SEARCH_PROTOCOL = NMEA_TEXT, // 3
SIRF_BINARY         = 4,
INIT_SEARCH_PROTOCOL = SIRF_BINARY, ///< 4 max value, goes through each_
→until we hit AUTODETECT
UNKNOWN             = 0xFF
} enumGPSProtocol;

```

2. In `driverGPSAllEntrance.c`, add this new protocol in `SetGpsProtocol()`. After this, the new protocol can be set in Aceinna Navigation Studio Web GUI.

```

BOOL SetGpsProtocol(int protocol, int fApply)
{
    switch(protocol)
    {
        case NMEA_TEXT:
        case NOVATEL_BINARY:
        case UBLOX_BINARY:
            break;
        default:
            return FALSE;
    }
    if(fApply)
    {
        gGpsDataPtr->GPSProtocol = protocol;
    }

    return TRUE;
}

```

3. In `driverGPS.c`, call the routine to decode this protocol.

```

switch(GPSData->GPSProtocol) {
    case NMEA_TEXT:
        parseNMEAAMessage(tmp, gpsMsg, GPSData);
        break;
    case NOVATEL_BINARY:
        parseNovotelBinaryMessage(tmp, gpsMsg, GPSData);
        break;
    case UBLOX_BINARY:
        parseUbloBinaryMessage(tmp, gpsMsg, GPSData);
        break;
    default:
        break;
}
}

```

4. Implement the decoding routine (`parseUbloBinaryMessage()`) in a proper file. For this example, it is implemented `processUbloxGPS.c`.

#### 4.4.3 The Definition of The Default Output Packet of The INS App

In the section *Get and Visualize the Output*, we can get INS app output data via the Python driver. The Python driver receives output from the unit, decodes the output packets and then feed decoded results to the Web GUI. If you want

to decode the output by yourself, you need to know the structure of the output packet, which is detailed in *OpenIMU UART Messaging*. The default INS app output packet type is “e2”, and it is defined in the following two tables.

('e2' = 0x6532)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6532	123		<CRC (U2)>

Payload:

Byte Offset	Name	Format	Notes
0	System Timer of sensors sampling	U4	LSB First msec
4	Above timer converted to a double type	D	LSB First second
12	Roll	F4	LSB First deg
16	Pitch	F4	LSB First deg
20	Yaw	F4	LSB First deg
24	X acceleration	F4	LSB First g
28	Y acceleration	F4	LSB First g
32	Z acceleration	F4	LSB First g
36	X acceleration bias	F4	LSB First m/s/s
40	Y acceleration bias	F4	LSB First m/s/s
44	Z acceleration bias	F4	LSB First m/s/s
48	X gyro	F4	
<b>4.4. GPS/INS App</b>			LSB First deg/s
52	Y gyro	F4	

#### 4.4.4 Synchronization to One PPS GPS Signal

The OpenIMU300 has the ability to synchronize a One PPS signal provided by the GPS receiver. The first step in the process is to connect the signal to the correct input pin on the OpenIMU300. In this case, Pin 2 serves as the input as described in [Connector Pinout](#).

See [synchronization to external clock signals](#) for more information on how to use the 1 PPS synchronization signal.

#### 4.4.5 About the GNSS/INS Fusion Algorithm

In the INS app, an 16-state extended Kalman filter is implemented to process measurements from a GPS receiver and an IMU unit. If you want to know more details about the algorithm, please refer to [EKF Algorithm](#).

---

**Note:** If you have any question, please search or post a new topic on [Aceinna Forum](#).

---

---

<sup>1</sup> Operation mode of the algorithm. 0 for waiting for the system to stabilize, 1 for initializing attitude, 2 and 3 for VG/AHRS mode, and 4 for INS mode. Please refer to the source code for details.

<sup>2</sup> 0 if linear acceleration is detected, 1 if no linear acceleration. Please refer to the source code for details.

<sup>3</sup> Indicate if the filtered yaw rate exceeds the turn switch threshold. 1 yes, 0 no. Please refer to the source code for details.

---

## Tutorial - What The User Needs to Know to Build The First Application

---

### OpenIMU Core

The OpenIMU Core is the foundation for the Platform application and all other example and custom applications. However, it is not supplied as a separate application. The OpenIMU Core provides the Board Support Package (BSP), FreeRTOS, command line interface capability, filters, GPS interface capability, math functionality, and various utilities, including the base examples for the C-language main function and the data acquisition functionality.

### EZ Embed Example Applications

The following applications are implicitly based on use of an *EZ Embed*\* OpenIMU units, such as the OpenIMU300ZI and OpenIMU330BI.

To get you acquainted with the OpenIMU environment, let's walk through the development of the following applications:

#### IMU Application

The term Inertial Measurement Unit (IMU) refers to a device that returns calibrated inertial-sensor data. This application forms the backbone of all other example applications as each requires inertial measurements to generate other results.

#### Static-Leveler Application

The Static-Leveler application uses accelerometer readings to measure the local gravity-field and compute the two-axis attitude (roll and pitch angles) of a body relative to the local-level frame. A Leveler application could be used to provide stabilization for cameras and other systems that require linear and rotational stability.

#### VG&AHRS Applications

The Vertical Gyro (VG) application and the Attitude and Heading Reference System (AHRS) application use rate-sensors, accelerometers, and (for the AHRS application) magnetometers to compute the attitude and heading of a body in space. Rate-sensors are used to propagate the attitude forward in time at high output data-rates (ODR) while accelerometers and magnetometers act as references, correcting for rate-sensor biases and attitude errors.

#### INS Application

The Inertial Navigation System (INS) application supports all of the features and operating modes of the VG&AHRS applications. In addition it includes the additional capability of interfacing with an external GPS receiver and associated software running on the processor for computation of navigation position information as well as orientation information.

### Robust CAN Example Applications

The CAN example applications are implemented for OpenIMU300RI unit with CAN interface. Next example applications available for OpenIMU300RI unit:

- IMU application which are using *SAE J1939 Messaging Standard*.
- VG\_AHRS application which are using *SAE J1939 Messaging Standard*.
- INS application which are using *SAE J1939 Messaging Standard*.

## 5.1 OpenIMU Core Details

All of the example applications and any custom applications are based on the OpenIMU Core firmware. The elements provided by the OpenIMU Core that are available to all example applications are as follows:

- Board Support Package (BSP) and FreeRTOS
- Default Pre-Filtering and Calibration Functions
- Default Data Acquisition Functions
- Default Message Functions
- Default Serial Debugging Functions
- Bootloader
- Python-Based Message Decoder
- Data Capture Functions Supporting the Aceinna Navigation Studio

Details of those elements are described in the following pages.

### 5.1.1 FreeRTOS & Board Support Package

#### FreeRTOS

The applications for all OpenIMU300 units use the FreeRTOS Real-Time Operating System (<https://www.freertos.org>), while OpenIMU330 units uses a simple real-time scheduler. FreeRTOS is very widely used, as it is feature-rich, has a small footprint, and can be used in commercial application without having to expose intellectual property.

FreeRTOS is licensed under the MIT Open Source License (<https://www.freertos.org/a00114.html>).

The critical feature of FreeRTOS:

- Scheduling Options
  - Pre-emptive
  - Co-operative
  - Round robin with time slicing
- Fast task notifications

- Configurable & scalable with a 6K to 12K ROM footprint
- Mutexes & semaphores
  - Mutexes with priority inheritance
  - Recursive mutexes
  - Binary and counting semaphores
- Chip and compiler agnostic
- Very efficient software timers
- Can be configured to never completely disable interrupts
- Easy to use API
- Easy to use message passing

**Board Support Package** - To Be Provided

### 5.1.2 Default Pre-Filtering and Calibration Functions

Several built-in digital filters are available to the user to provide additional filtering. In particular, a selection of second-order Butterworth low-pass filters are provided. Butterworth filters were chosen for their maximally flat passband and straight-forward frequency responses. Available cutoff-frequencies are:

- 50 Hz
- 40 Hz
- 25 Hz
- 20 Hz
- 10 Hz
- 5 Hz
- 2 Hz
- 0 Hz (Unfiltered)

In the firmware, these filter are implemented using fixed-point math (which operate on sensor counts, not floating-point values). This was done to take advantage of the speed associated with integer-math operations.

Built-in filters are selected in several different ways:

1. Cutoff frequencies can be set in the default user-configuration structure, *UserConfigurationStruct*. This is the approach taken in this example.
2. The configuration can be changed (either temporarily or permanently) using the Aceinna Navigation Studio interface.
3. Commands can be sent to the unit over the serial interface. This enables the cutoff frequency to be changed during operation, if desired.

#### **Calibration:**

Once filtered, the OpenIMU firmware then applies calibration data to the sensor counts, compensating for temperature-related bias effects, sensor scale-factors, and misalignment.

### 5.1.3 Default Data Acquisition Functions

#### Contents

- *Acquiring Sensor Data*

OpenIMU makes data-acquisition simple by reducing the steps required to get high-quality, inertial sensor data. Sensor drivers, filtering, and calibration are handled without the need for additional user input.

The main routine controlling sensor sampling and processing is *TaskDataAcquisition*. This task calls the routines that acquire sensor measurements, filter the data, and apply calibration. In particular, the task calls the following, which provides functions to acquire sensor data:

```
inertialAndPositionDataProcessing(dacqRate);
```

After completion of the sensor processing steps, it then calls the algorithm that operates on sensor readings to create processed output.

#### Acquiring Sensor Data

Inside *inertialAndPositionDataProcessing()* several getter-functions are provided. These functions obtain sensor data directly from the sensor data-buffers. Function names, described in the following table, were chosen to make the task of each function clear.

Table 1: Sensor Measurement Getter Functions

Getter Function	Description	Units
<i>GetAccelData_g()</i>	Obtain accelerometer data	[g]
<i>GetAccelData_mPerSecSq()</i>		[m/s <sup>2</sup> ]
<i>GetRateData_radPerSec()</i>	Obtain rate-sensor data	[r/s]
<i>GetRateData_degPerSec()</i>		[°/s]
<i>GetMagData_G()</i>	Obtain magnetometer data	[G]
<i>GetBoardTempData()</i>	Obtain temperature data	[°C]

**Note:** Most inertial algorithm development will use [m/s<sup>2</sup>], [r/s], and [G]. However getters that provide accelerometer and rate-sensor data in [g] and [°/s] are also available for the designer who chooses to work in these units.

These getters work by populating the array whose address is provided as an argument to the function. In this example, the functions load the data directly into the data-structure elements *gIMU.accel\_g*, *gIMU.rate\_degPerSec*, *gIMU.mag\_G*, and *gIMU.temp\_C*.

**Note:** Structure elements (*accel\_g*, *rate\_degPerSec*, etc.) are all defined as doubles in the data structure created in *UserMessaging.h*. This is done to match the datatype required by the getter functions, described above.

```
// IMU data structure
typedef struct {
    // Timer output counter
    uint32_t timerCnt, dTimerCnt;

    // Algorithm states
```

(continues on next page)



(continued from previous page)

```

double accel_g[3];
double rate_degPerSec[3];
double mag_G[3];
double temp_C;
} IMUDataStruct;

extern IMUDataStruct gIMU;

```

### 5.1.4 Default Message Functions

#### Contents

- *Serial Message Definition*
- *UserMessaging.h Modifications*
- *UserMessaging.c Modifications*
- *Default Configuration Settings*
- *Testing using Serial Terminal Emulator*

#### Serial Message Definition

A streaming, serial message can be generated by the OpenIMU platform. In this example, a message matching the requirements, defined earlier, is created. It consists of:

1. An integer counter, representing time in  $[ms]$
2. A floating-point representation of time, in  $[s]$
3. Accelerometer readings, in  $[g]$
4. Rate-Sensor readings, in  $[^{\circ}/s]$
5. Magnetometer readings, in  $[G]$
6. Board temperature, in  $[^{\circ}C]$

To generate this output, a serial-message was created in *UserMessaging.c* and *UserMessaging.h*. In the firmware, the message is given the name, *USR\_OUT\_SCALED1*, along with the packet code “s1” (with lower-case S representing scaled).

To form the message, the first step is to define the message components and determine the total number of bytes the message will occupy. The components of the message, variable type, and number of bytes are listed in the following table:

Table 2: User-Defined Serial Message Components

Message Component	Description		Number of Variables	Total Bytes
	Type	Bytes		
Integer counter	uint32_t	4	1	4
Time variable	double	8	1	8
Accelerometer Readings (3 axis)	float	4	3	12
Rate-Sensor Readings (3 axis)	float	4	3	12
Magnetometer Readings (3 axis)	float	4	3	12
Board-Temperature Readings (3 axis)	float	4	1	4

This shows that the *payload* section of the output message (not including preamble, message type, or CRC) consists of 52 bytes.

Adding this message to the firmware requires modifications to two files: *UserMessaging.c* and *UserMessaging.h*.

### ***UserMessaging.h* Modifications**

The packet code and number of bytes must be added to *UserMessaging.h*. This requires adding the output packet code to the packet-type enum variable:

```
// User output packet codes, change at will
typedef enum {
    USR_OUT_NONE = 0,    // 0
    USR_OUT_TEST,        // 1
    USR_OUT_DATA1,       // 2
    USR_OUT_DATA2,       // 3
    // add new output packet type here, before USR_OUT_MAX
    USR_OUT_SCALED1,     // 4
    USR_OUT_MAX
} UserOutPacketType;
```

and creating a *#define* identifier to hold the payload length

```
#define USR_OUT_SCALED1_PAYLOAD_LEN (52)
```

These can be found in the IMU example code.

## UserMessaging.c Modifications

With the above additions to *UserMessaging.h* made, the output message can be added to *UserMessaging.c*, completing the process. To accomplish this, add a new case to the switch-statement found in *HandleUserOutputPacket()* using the output name added to *UserMessaging.h*:

```
case USR_OUT_SCALED1:
{
    // The payload length (NumOfBytes) is based on the following:
    // 1 uint32_t (4 bytes) = 4 bytes
    // 1 double (8 bytes) = 8 bytes
    // 3 floats (4 bytes) = 12 bytes
    // 3 floats (4 bytes) = 12 bytes
    // 3 floats (4 bytes) = 12 bytes
    // 1 floats (4 bytes) = 4 bytes
    // =====
    //          NumOfBytes = 52 bytes
    *payloadLen = USR_OUT_LEV1_PAYLOAD_LEN;

    // Output time as represented by gIMU.timerCntr (uint32_t
    // incremented at each call of the algorithm)
    uint32_t *algoData_1 = (uint32_t*)(payload);
    *algoData_1++ = gIMU.timerCntr;

    // Output a double representation of time generated from
    // gLeveler.itow
    double *algoData_2 = (double*)(algoData_1);
    *algoData_2++ = 1.0e-3 * (double)(gIMU.timerCntr);

    // Set the pointer of the sensor array to the payload
    float *algoData_3 = (float*)(algoData_2);
    *algoData_3++ = (float)gIMU.accel_g[X_AXIS];
    *algoData_3++ = (float)gIMU.accel_g[Y_AXIS];
    *algoData_3++ = (float)gIMU.accel_g[Z_AXIS];

    *algoData_3++ = (float)gIMU.rate_degPerSec[X_AXIS];
    *algoData_3++ = (float)gIMU.rate_degPerSec[Y_AXIS];
    *algoData_3++ = (float)gIMU.rate_degPerSec[Z_AXIS];

    *algoData_3++ = (float)gIMU.mag_G[X_AXIS];
    *algoData_3++ = (float)gIMU.mag_G[Y_AXIS];
    *algoData_3++ = (float)gIMU.mag_G[Z_AXIS];

    *algoData_3++ = (float)gIMU.temp_C;
}
break;
```

Data is appended to the payload array using pointers. This enables variables of different datatypes to fit into the payload array (defined as an array of 8-bit unsigned integers); this approach is highlighted in the previous code snippet and is done by generating a pointer of the desired type to a typecast version of the payload address. In the example above, 32-bit unsigned integer data is appended to the payload, followed by double and floating-point variables.

Finally, the packet type must be added to the switch-statement in *setUserPacketType()* to allow the firmware to select the packet:

```
case USR_OUT_SCALED1:          // packet with arbitrary data
    _outputPacketType = type;
    _userPayloadLen = USR_OUT_SCALED1_PAYLOAD_LEN;
```

(continues on next page)

(continued from previous page)

```
break;
```

and the packet-code must be added to the list of user output packets, *userOutputPackets*.

```
// packet codes here should be unique -
// should not overlap codes for input packets and system packets
// First byte of Packet code should have value >= 0x61
usr_packet_t userOutputPackets[] = {
//   Packet Type           Packet Code
  {USR_OUT_NONE,          {0x00, 0x00}},
  {USR_OUT_TEST,          "zT"},
  {USR_OUT_DATA1,         "z1"},
  {USR_OUT_DATA2,         "z2"},
// place new type and code here
  {USR_OUT_SCALED1,       "s1"},
  {USR_OUT_MAX,           {0xff, 0xff}}, // ""
};
```

These changes are found in *UserMessaging.c*.

## Default Configuration Settings

To make the “s1” serial message (created previously) the default output, make changes to the default user-configuration structure found in *UserConfiguration.c*:

```
// Default user configuration structure
// Saved into EEPROM of first startup after reloading the code
// or as a result of processing "rD" command
// Do Not remove - just add extra parameters if needed
// Change default settings if desired
const UserConfigurationStruct gDefaultUserConfig = {
    .dataCRC           = 0,
    .dataSize          = sizeof(UserConfigurationStruct),
    .userUartBaudRate  = 115200,
    .userPacketType    = "s1",
    .userPacketRate    = 10,
    .lpfAccelFilterFreq = 25,
    .lpfRateFilterFreq = 25,
    .orientation       = "+X+Y+Z"
    // add default parameter values here, if desired
};
```

**Note:** *userPacketType* was set to “s1” to cause the new packet to be broadcast by default. Additionally, the desired message baud rate and message rate are set to 115.2 kbps and 10 [Hz], respectively. Finally, the accelerometer and rate-sensor filters are set to 25 Hz.

## Testing using Serial Terminal Emulator

At this point, the IMU application has been implemented and the output messaging created. Build and upload the firmware to the OpenIMU. A serial terminal (such as TeraTerm) can be used to verify if a message is being generated by the device. In the following figure, output messaging creation can be verified by searching for the string “UUs1”.

If present, the message is being generated; whether the message is populated correctly requires the use of additional tools.

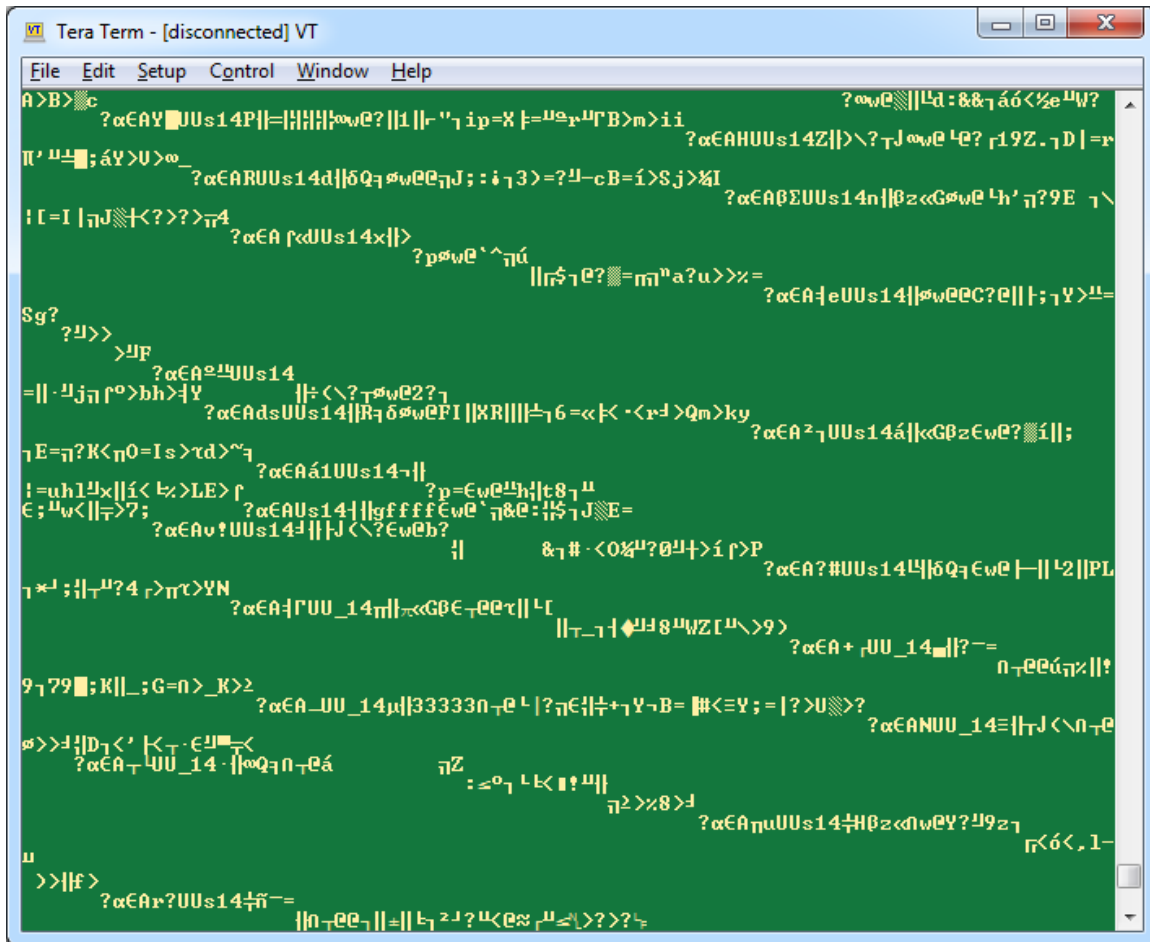


Fig. 1: Test of Serial Message Output

**Note:** In the above figure the message preamble sometimes displays as “UU\_1”. This is solely a TeraTerm glitch. Other serial terminal programs (such as CoolTerm) do not show such behavior.

### 5.1.5 Default Serial Debugging Functions

#### Contents

- *Generating Debug Messages*
- *Compile and Test*

#### Generating Debug Messages

##### Creating the Message:

#### 5.1. OpenIMU Core Details

Debug messages, using the built-in debugging capability of the OpenIMU platform, are added to the IMU application to verify that the firmware obtains the correct sensor reading; the complete implementation is found in *dataProcessingAndPresentation.c* in the IMU application code. The relevant debugger calls are:

```
DebugPrintFloat("Time: ", 0.001 * gIMU.timerCntr, 3);
DebugPrintFloat(" , AccelZ: ", gIMU.accel_g[Z_AXIS], 3);
DebugPrintFloat(" , RateZ: ", gIMU.rate_degPerSec[Z_AXIS], 3);
DebugPrintFloat(" , MagX: ", gIMU.mag_G[X_AXIS], 3);
DebugPrintFloat(" , Temp: ", gIMU.temp_C, 2);
DebugPrintEndline();
```

In the output message, z-axis acceleration and rate-sensor measurements, provided in  $[g]$  and  $[^{\circ}/s]$ , are obtained along with x-axis magnetic-field readings (in  $[G]$ ) and board-temperature (in  $[^{\circ}C]$ ). This subset of sensor information is selected to test the output of all sensors, while keeping the size of the debug message small.

Arguments to *DebugPrintFloat()* consist of:

1. A character-string describing the output message
2. The floating-point value to be output
3. The number of significant digits in the output message

In this example, only *DebugPrintFloat()* is used to output a debug message, other debug message functions are available. In particular, the following messages (provided in *debug.c*) form the complete list:

```
DebugPrintString();
DebugPrintInt();
DebugPrintLongInt();
DebugPrintHex();
DebugPrintFloat();
DebugPrintEndline();
```

## Compile and Test

The final step is to build and upload the firmware to the OpenIMU hardware using the PIO framework. When complete, use a terminal program (such as TeraTerm in Windows) to connect to the appropriate COM port to assess if the program is operating as expected.

### Debug Communication Settings:

Debug messages are provided as serial messages over the third port of the OpenIMU platform. When connected to a PC, the device generates four COM ports. In this case, the ports are 40, 41, 42, and 43. The first COM port is the serial messaging port (discussed in the [Platform Communications](#) section), the second port can be used for serial inputs to the platform (such as GPS), and the fourth is unconnected.

The nominal serial baud-rate setting is 38.4 kbps. This can be set to other rates, such as 57.6 kbps, 115.2 kbps, or 230.4 kbps via the argument to *InitDebugSerialCommunication()*, found in *main.c*. For the IMU application, this value was changed to 230.4 kbps.

### System Testing using Debug Communications:

To test the OpenIMU output, perform the following:

1. Place the unit on a level table top
2. With the unit sitting flat, the z-axis acceleration will be close to  $-1.0 [g]$
3. Rotate the unit clockwise (about the positive z-axis) to generate a positive z-axis angular-rate

4. Orient the unit so the y-axis is aligned with magnetic-north. This results in an x-axis magnetic-field reading close to zero [G]. Orienting the unit's x-axis in any other compass direction will result in a non-zero magnetic-field reading that increases until the axis is pointed along the north/south direction, at which it reaches its maximum value.
5. Temperature readings reflect values slightly higher than the ambient temperature, as the readings reflect the temperature of the electronics.

The results of these statements are found in the following figure:

Fig. 2: IMU Debug Output

This output provides confidence that the IMU is obtaining the correct sensor measurements.

### Suggested Operation

During normal operations, when using the OpenIMU in your system, it is best to disable the debug output. This will reduce the load on the platform and free up the processing capability for other tasks.

## 5.1.6 Bootloader

Each of the examples have its associated application pre-built as .bin files, which can be downloaded directly onto the OpenIMU hardware directly from [Aceinna Navigation Studio](#).

### Download Procedure

### Connect to the OpenIMU Python Server

From the terminal window, issue the command to start the OpenIMU Python Server:

```
C:\Users\labuser\Documents\platformio\python-openimu>python server.py
autoconnected
Connected ....OpenIMU300ZA5020-3885-01 1.0.1 SN:1710000008
```

Fig. 3: Python Server Connection

### Connect the Unit to the Aceinna Navigation Studio

From the [Aceinna Navigation Studio](#) main page, select *Code* and *Apps* from the menu on the left-hand side of the window

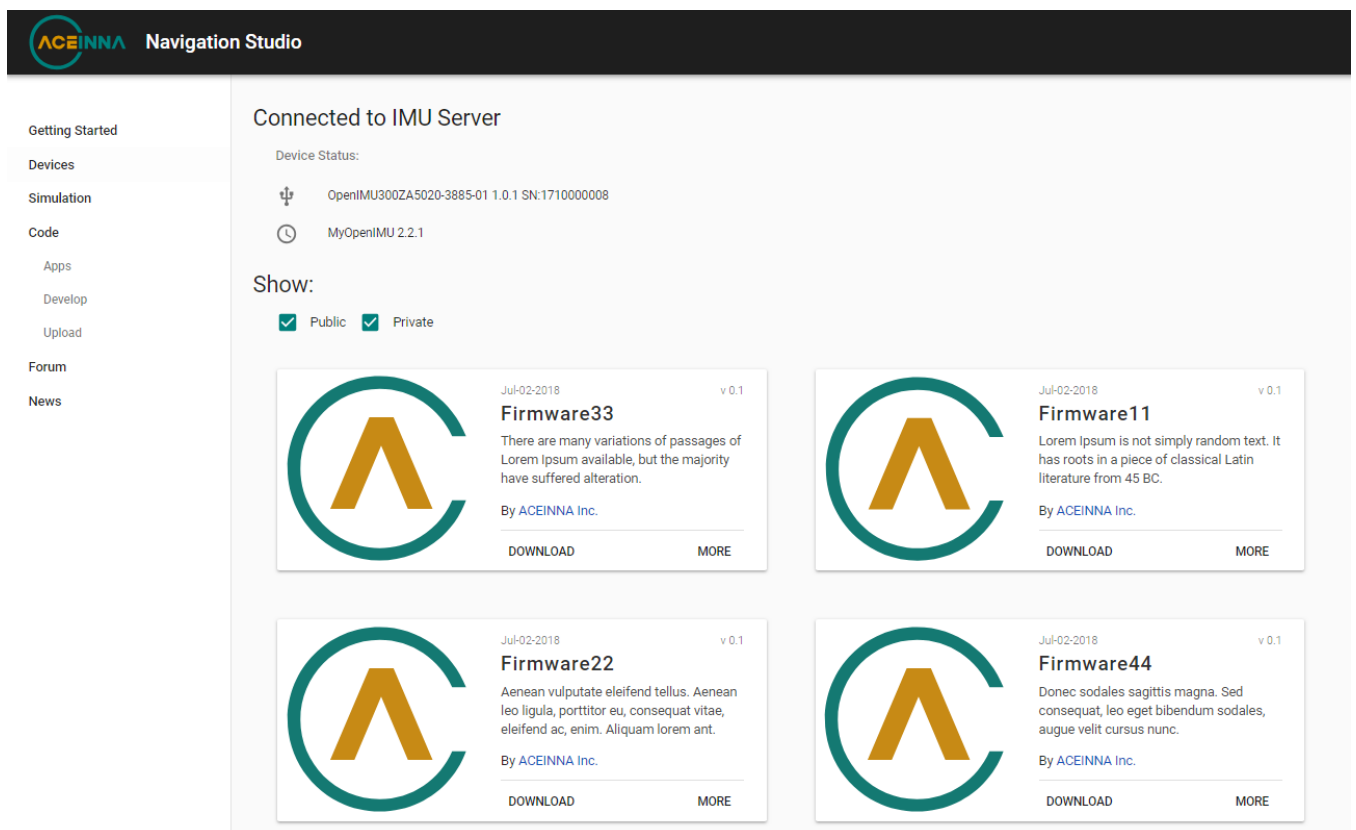


Fig. 4: ANS Applications

### Downloading the Application

Navigate to the desired application and click the *Download* link at the bottom of the application box. In this case, select the **IMU Application Download** link.

Once the *Download* link has been clicked, a progress bar at the top of the application box will indicate how much time is left to download the application:

### Download Progress View In Terminal

Additionally terminal messages in the window in which the Python server is running will indicate progress. Once complete, the terminal will indicate *Success* and *restart the app*. At this point the unit is now running the downloaded



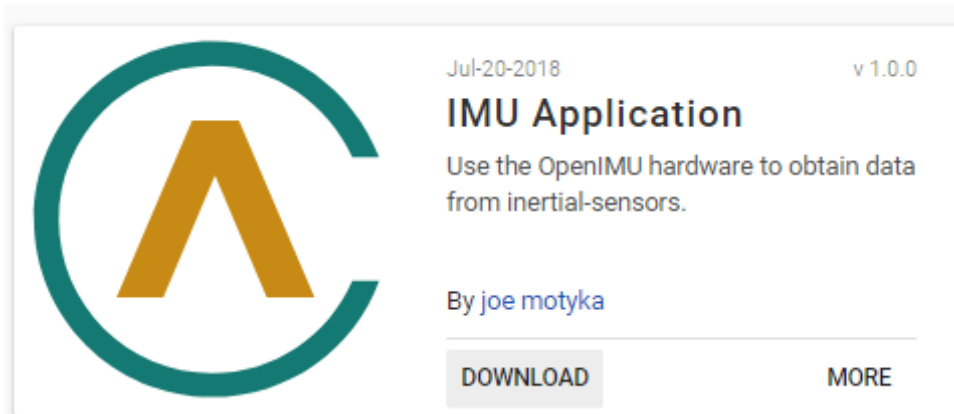


Fig. 5: *IMU Application*

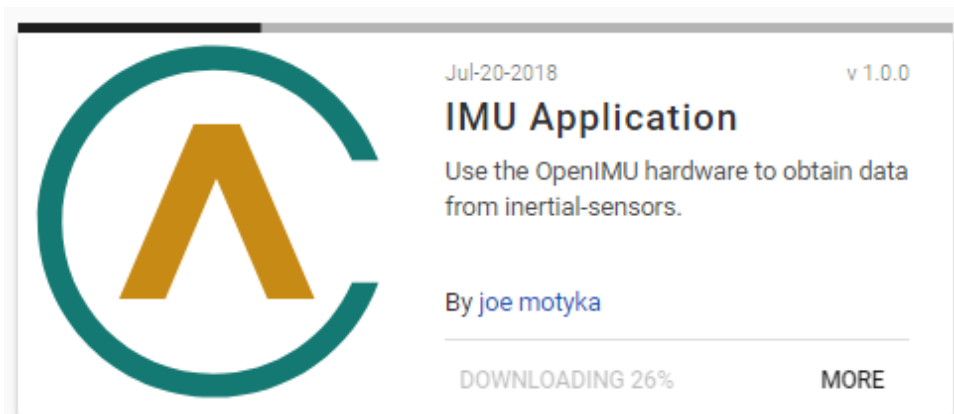


Fig. 6: *Application Download Progress*

application.

```
(240, 138240)
Success
(240, 138480)
Success
(240, 138720)
Success
(240, 138960)
Success
(240, 139200)
Success
(240, 139440)
Success
(240, 139680)
Success
(240, 139920)
Success
(240, 140160)
Success
(240, 140400)
Success
(240, 140640)
Success
(240, 140880)
Success
(240, 141120)
Success
(240, 141360)
Success
(240, 141600)
Success
(240, 141840)
Success
(240, 142080)
Success
(240, 142320)
Success
(24, 142560)
Success
Restarting app ...
autoconnected
Connected ....OpenIMU300ZA5020-3885-01 1.0.1 SN:1710000008
```

Fig. 7: Terminal Download Progress Screen

The unit can now be connected to the Navigation Studio and data plotted or saved in an output file.

### 5.1.7 Python-Based Message Decoder

#### Contents

- *Creating a python-based decoder*

#### Creating a python-based decoder

The first step to using the OpenIMU decoder and spooling tools, *python-openimu*, to properly decode an output message, is to define the message in the file *openimu.json*. For the “s1” message, the following is added to the file:

```

{
  "name": "sl",
  "description": "IMU Scaled-Sensor Output Message",
  "payload": [{
    "type": "uint32",
    "name": "timeCntr",
    "unit": "msec"
  },
  {
    "type": "double",
    "name": "time",
    "unit": "s"
  },
  {
    "type": "float",
    "name": "xAccel",
    "unit": "g"
  },
  {
    "type": "float",
    "name": "yAccel",
    "unit": "g"
  },
  {
    "type": "float",
    "name": "zAccel",
    "unit": "g"
  },
  {
    "type": "float",
    "name": "xRate",
    "unit": "deg/s"
  },
  {
    "type": "float",
    "name": "yRate",
    "unit": "deg/s"
  },
  {
    "type": "float",
    "name": "zRate",
    "unit": "deg/s"
  },
  {
    "type": "float",
    "name": "xMag",
    "unit": "G"
  },
  {
    "type": "float",
    "name": "yMag",
    "unit": "G"
  },
  {
    "type": "float",
    "name": "zMag",
    "unit": "G"
  }
  ]
}

```

(continues on next page)

(continued from previous page)

```

    },
    {
      "type": "float",
      "name": "temp",
      "unit": "degC"
    }
  ],
  "graphs": [{
    "name": "Acceleration",
    "units": "g",
    "xAxis": "Time (s)",
    "yAxes": ["xAccel", "yAccel", "zAccel"],
    "colors": ["#FF0000", "#00FF00", "#0000FF"],
    "yMax": 5
  },
  {
    "name": "Angular-Rate",
    "units": "deg/s",
    "xAxis": "Time (s)",
    "yAxes": ["xRate", "yRate", "zRate"],
    "colors": ["#FF0000", "#00FF00", "#0000FF"],
    "yMax": 200
  },
  {
    "name": "Magnetic-Field",
    "units": "G",
    "xAxis": "Time (s)",
    "yAxes": ["xMag", "yMag", "zMag"],
    "colors": ["#FF0000", "#00FF00", "#0000FF"],
    "yMax": 5
  },
  {
    "name": "Board-Temperature",
    "units": "degC",
    "xAxis": "Time (s)",
    "yAxes": ["temp"],
    "colors": ["#FF0000"],
    "yMax": 100
  }
  ]
}

```

This information tells the decoder the order of the output data in the serial message, its type (float, double, int, etc.), as well as the units associated with the data. It also defines how the data should be plotted, including axis-titles and colors.

**Note:** A useful tool to check if the json-file is properly formatted is found at: <https://jsonlint.com>

## 5.1.8 Data Capture Functions Supporting the Aceinna Navigation Studio

### Contents

- *OpenIMU Server*
- *Connect to Aceinna Navigation Studio*
- *Displaying Data*
- *Logging Data*

Capturing, Displaying, and Saving Data Using the Aceinna Navigation Studio

With the following complete:

1. Serial output-message created and running on the OpenIMU hardware
2. The message description added to *openimu.json*
3. *python-openimu* installed on your system

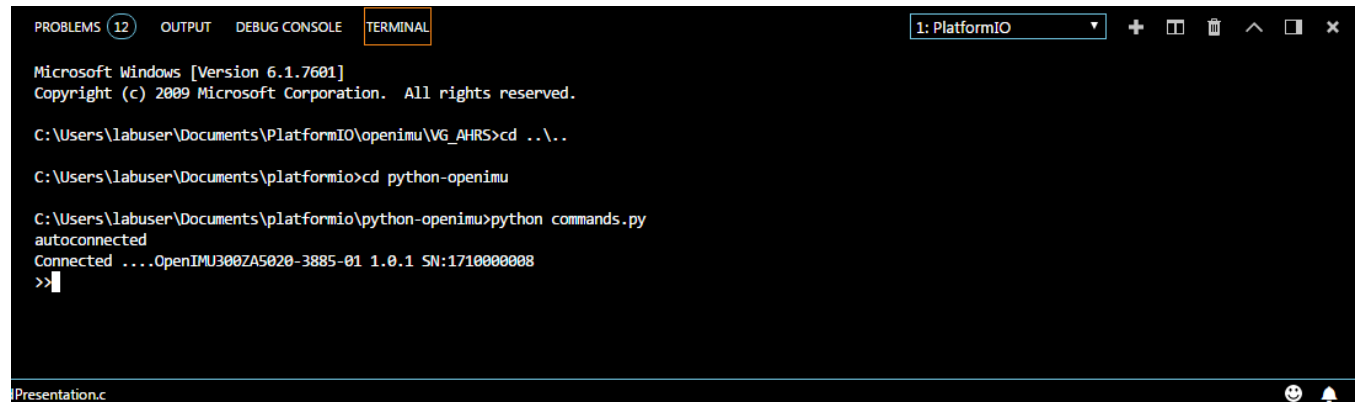
you are now ready to collect IMU data.

## OpenIMU Server

To capture data using the [Aceinna Navigation Studio](#), the first step is to start the python-based server that will capture the serial data streaming over the COM port. This can be done by sending the following command at a terminal prompt from the *python-openimu* folder:

```
python commands.py
```

This initiates a search for the OpenIMU device on the machine's COM ports. When detected, the terminal returns a message similar to the following:



```

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\labuser\Documents\PlatformIO\openimu\VG_AHRS>cd ../../
C:\Users\labuser\Documents\platformio>cd python-openimu
C:\Users\labuser\Documents\platformio\python-openimu>python commands.py
autoconnected
Connected ...OpenIMU300ZA5020-3885-01 1.0.1 SN:1710000008
>>

```

Fig. 8: Server-Connection Message at the Terminal Prompt

Once connected to the IMU type 'start\_server' to start the server. More instructions on the Python driver are found [here](#)

## Connect to Aceinna Navigation Studio

To capture and display data on the [Aceinna Navigation Studio](#), open a browser to <https://developers.aceinna.com> and log in. From the menu on the left, select *Devices* and *Connect*. The following will appear if connected properly:

If desired, the packet output rate and other settings can be changed here.

After connecting to the OpenIMU device, the terminal reflects this by displaying the configuration of the unit:

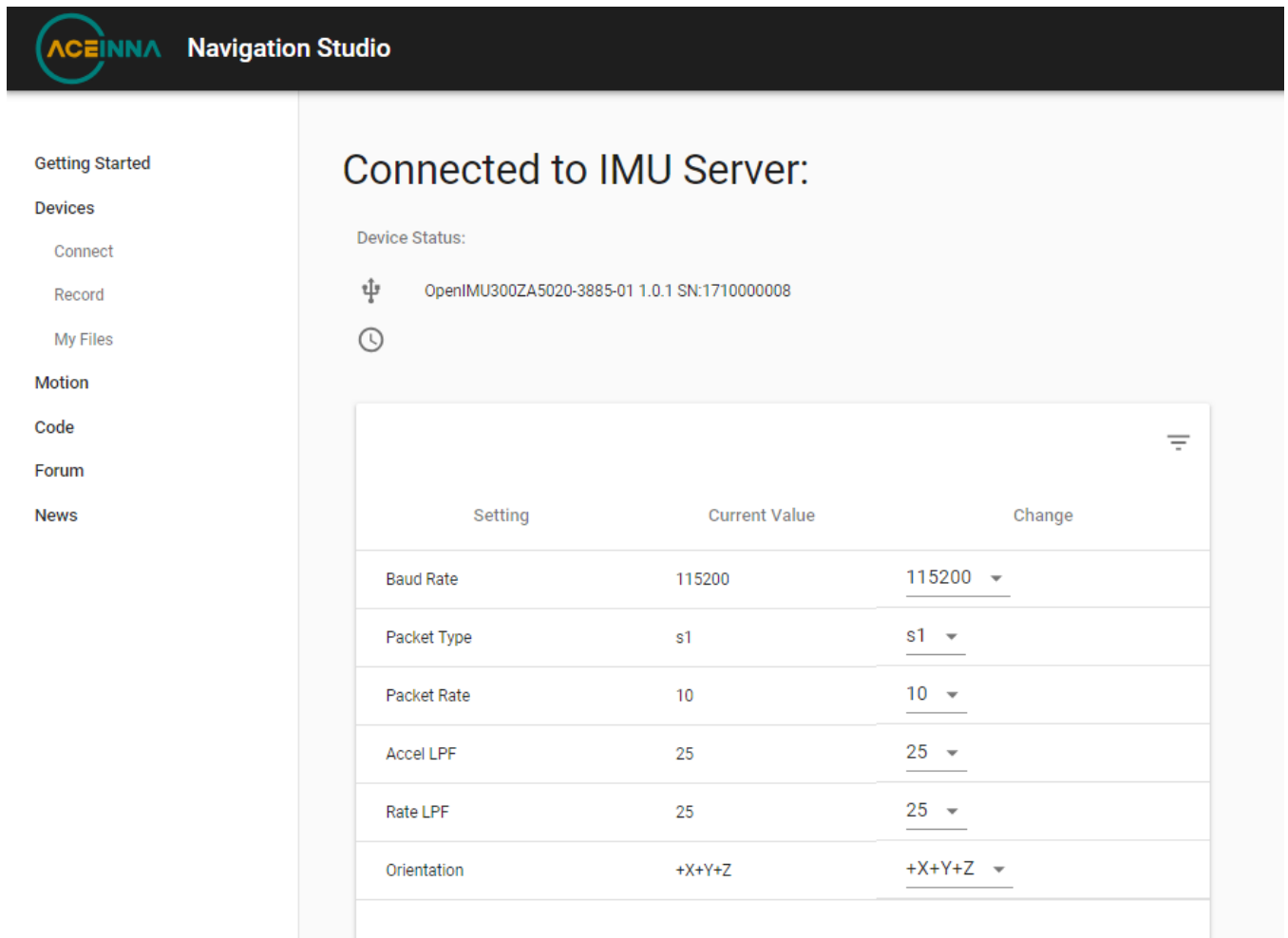


Fig. 9: Connection to IMU Server

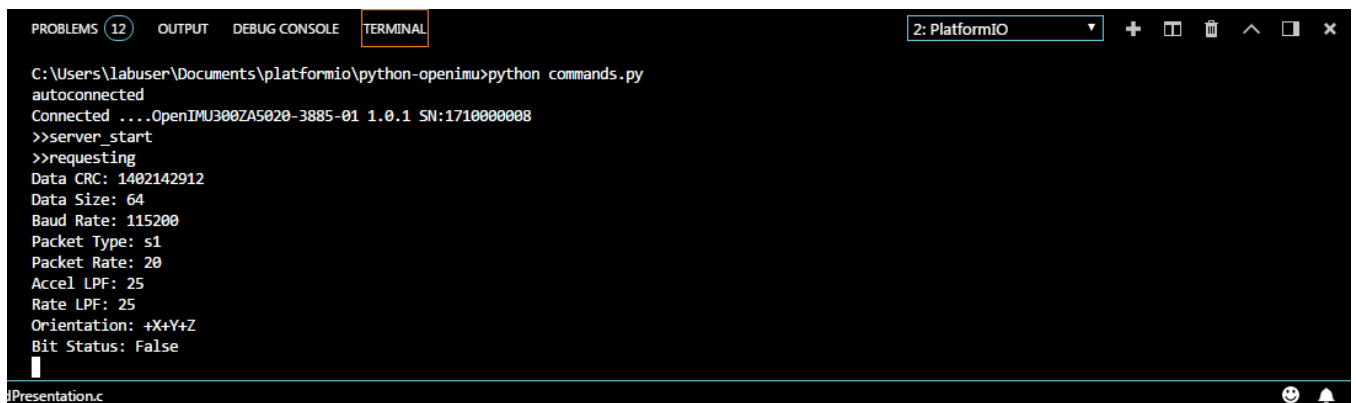


Fig. 10: Server-Connection Message at the Terminal Prompt

## Displaying Data

For a live display of data from the device, select the *Record* menu then click on the *Play* button. An example capture of the accelerometer data follows:

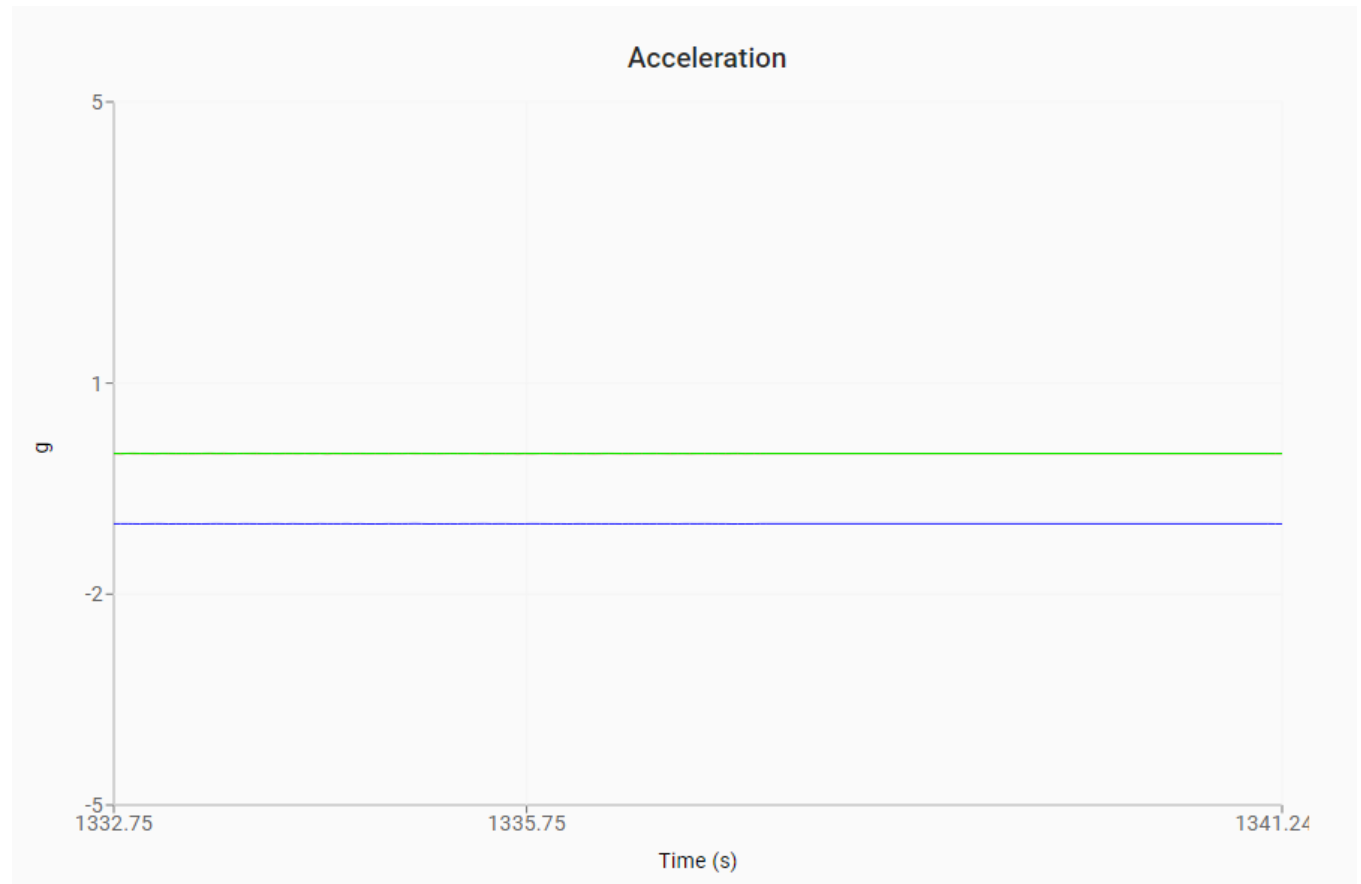


Fig. 11: Plot of IMU Accelerometer Data

## Logging Data

To log data select the *Log Control* switch. The output file consists of data found in the serial message. In particular the message consists of:

- Time (in counts and seconds)
- Accelerometer data (in  $[g]$ )
- Rate-Sensor data (in  $[^{\circ}/s]$ )
- Magnetometer data (in  $[G]$ )
- Board-Temperature data (in  $[^{\circ}C]$ )

The following figure shows the contents of the captured data file, indicating that all selected data are saved as intended.

```

timeITOW,time,xAccel,yAccel,zAccel,xRate,yRate,zRate,xMag,yMag,zMag,temp
292300,292.300000000000,-0.00201910,0.00070894,-0.99919415,0.04940020,0.01053898,-0.08763175,0.22584580,0.27953801,0.57122874,29.01953125
292320,292.320000000000,-0.00286614,0.00054671,-1.00162399,0.02000007,-0.06367743,0.03553406,0.22737505,0.28003392,0.57281804,29.01953125
292330,292.330000000000,-0.00241628,0.00012953,-1.00057185,0.01204931,-0.02115907,0.00998531,0.22803544,0.28025734,0.57355219,29.01953125
292350,292.350000000000,-0.00190265,0.00022255,-1.00088942,0.02340962,-0.09668475,-0.03467730,0.22872530,0.28053418,0.57375592,29.01953125
292360,292.360000000000,-0.00168020,0.00009613,-1.00063181,0.03895983,-0.12499246,-0.05772449,0.22890468,0.28092027,0.57357854,29.01953125
292380,292.380000000000,-0.00097376,-0.00106805,-1.00111389,-0.01535512,-0.08603050,0.00486011,0.22931509,0.28193080,0.57333028,29.01953125
292400,292.400000000000,-0.00138218,-0.00092208,-1.00077891,0.03456460,-0.02484225,0.01820971,0.22962117,0.28328615,0.57350957,29.01953125
292410,292.410000000000,-0.00039054,-0.00023197,-1.00058722,0.06906004,-0.04079971,0.04235101,0.22986402,0.28346157,0.57377213,29.01953125
292430,292.430000000000,-0.00236613,-0.00039699,-1.00098121,0.05435806,-0.09547580,-0.02284101,0.22983892,0.28266230,0.57405126,29.01953125
292440,292.440000000000,-0.00274126,-0.00040483,-1.00076270,0.02944281,-0.03539148,-0.03668196,0.22979984,0.28212029,0.57406676,29.01953125
292460,292.460000000000,-0.00191691,-0.00052903,-0.99982393,0.04298836,-0.08904774,0.04402783,0.22965389,0.28181332,0.57433248,29.02343750
292480,292.480000000000,-0.00237940,0.00018120,-1.00157332,0.01357799,-0.00926387,0.01965515,0.22877936,0.28178373,0.57466304,29.02343750
292490,292.490000000000,-0.00193565,0.00057816,-1.00089025,0.04647730,-0.03253048,-0.03305940,0.22834855,0.28164941,0.57484525,29.02343750
292510,292.510000000000,-0.00170721,-0.00133564,-1.00110674,-0.00112826,-0.09231940,-0.04214227,0.22845493,0.28133366,0.57470071,29.02343750
292520,292.520000000000,-0.00204248,-0.00027414,-1.00078738,0.04349720,-0.10027670,-0.03038196,0.22872335,0.28112915,0.57476658,29.02343750
292540,292.540000000000,-0.00122875,-0.00086629,-1.00213861,0.11997440,-0.01112851,-0.00473972,0.22961204,0.28162318,0.57578170,29.02343750
292560,292.560000000000,-0.00224951,-0.00059281,-0.99990159,0.02699803,-0.06606586,0.02852671,0.22997279,0.28219634,0.57622135,29.02343750
292570,292.570000000000,-0.00189999,-0.00008876,-1.00032949,0.01991043,-0.06967562,0.03851629,0.23023081,0.28256884,0.57590139,29.02343750
292590,292.590000000000,-0.00156008,0.00074339,-0.99992335,-0.01364117,-0.00100617,-0.02615706,0.22976674,0.28272992,0.57516634,29.02343750
292600,292.600000000000,-0.00111164,0.00063984,-0.99843735,0.03249335,0.01456794,0.01296413,0.22928962,0.28280434,0.57512051,29.02343750
292620,292.620000000000,-0.00139861,0.00008168,-1.00039744,0.00778001,-0.03491977,-0.01149094,0.22820659,0.28288883,0.57494193,29.02343750
292640,292.640000000000,-0.00193238,0.00060333,-1.00168681,0.07713205,-0.09581902,-0.03092752,0.22721386,0.28198451,0.57343125,29.02343750
292650,292.650000000000,-0.00217579,0.00076254,-1.00140095,0.06470323,-0.07971172,0.02126878,0.22710972,0.28141144,0.57328844,29.02343750
292670,292.670000000000,-0.00261411,-0.00065658,-1.00232029,0.03621025,-0.04235784,0.04664550,0.22736281,0.28097749,0.57398975,29.02343750
292680,292.680000000000,-0.00143532,-0.00031316,-1.00080037,0.03886847,-0.04787493,0.06177394,0.22782773,0.28112459,0.57416064,29.02343750
292700,292.700000000000,-0.00205310,0.00010566,-1.00093722,-0.02246748,-0.09901554,-0.05674051,0.22903086,0.28176004,0.57414663,29.02343750
292720,292.720000000000,-0.00156667,-0.00042325,-1.00021029,-0.03647065,-0.07424885,0.03083275,0.23028445,0.28279892,0.57365668,29.02343750
292730,292.730000000000,-0.00160590,-0.00105158,-1.00117230,-0.04289529,-0.02480554,-0.00788887,0.23073351,0.28291312,0.57391602,29.02343750
292750,292.750000000000,-0.00261864,-0.00120951,-1.00016785,0.05224881,-0.09988170,-0.04751080,0.23110659,0.28316671,0.57447767,29.02343750
292760,292.760000000000,-0.00279064,-0.00152914,-1.00021803,0.04901344,-0.11903654,-0.01348535,0.23104802,0.28338522,0.57485086,29.02343750
292780,292.780000000000,-0.00288121,-0.00083522,-1.00257635,0.02901294,-0.06123649,-0.03690565,0.23012848,0.28313836,0.57615322,29.02343750
292800,292.800000000000,-0.00189324,-0.00032563,-1.00095844,-0.01422259,-0.06009072,-0.05412498,0.22963716,0.28219426,0.57603168,29.02343750
292810,292.810000000000,-0.00214101,-0.00097840,-1.00242209,-0.00227018,-0.03636564,-0.02940226,0.22945428,0.28176531,0.57571393,29.02343750
292830,292.830000000000,-0.00126435,-0.00099424,-0.99944186,0.02882084,-0.08648770,0.00064673,0.22906525,0.28121820,0.57582277,29.02343750
292840,292.840000000000,-0.00111330,-0.00057698,-0.99951899,-0.00411007,-0.05838403,0.03477932,0.22897796,0.28080669,0.57546842,29.02343750
292860,292.860000000000,-0.00135733,0.00011049,-1.00118256,0.02354836,-0.03794597,0.00050074,0.22919057,0.28011021,0.57530272,29.02343750
292880,292.880000000000,-0.00151269,0.00012875,-0.99990767,0.14041716,-0.02024340,0.00855396,0.22876379,0.28019711,0.57566289,29.02343750
292890,292.890000000000,-0.00238976,0.00022297,-1.00068378,0.00828160,-0.08348626,-0.00697362,0.22832575,0.28066689,0.57593662,29.02343750

```

Fig. 12: IMU Angle Data File

## 5.2 Inertial Measurement Unit (IMU) Application

The Inertial Measurement Unit (IMU) application enables the OpenIMU hardware to provide inertial-sensor data from accelerometers, rate-sensors, and magnetometers.

The exact combination of sensor data you use will depend upon the ultimate goal of your project. However, at least a subset of this data is required to create an application that estimates attitude, position, and/or heading.

The IMU application performs the following functions:

- Sets the default OpenIMU configuration for the IMU application
- Acquires Sensor Data - acceleration, angular-rate, local magnetic-field, and sensor temperature data
- Generates and sends the following output message to the UART:
  - A relative time measurement (both integer and decimal values)
  - Acceleration readings in  $[g]$
  - Rate-sensor readings in  $[^{\circ}/s]$
  - Magnetic-field readings in  $[G]$
  - Sensor temperature readings in  $[^{\circ}C]$

## 5.3 Static-Leveler Application

The static-leveler application enables the OpenIMU hardware to provide roll and pitch estimates (the angles that the x and y-axes are rotated away from level) using only accelerometer measurements. This simple example is based on the



*IMU Example Application*

The Static Leveler application performs the following functions:

- Sets the default OpenIMU configuration for the Leveler application
- Acquires Sensor Data - acceleration, angular-rate, local magnetic-field, and sensor temperature data
- Executes the Leveler application algorithms and other relevant math functions to create output data:
  - Compute the acceleration unit-vector.
  - Normalize using the magnetometer readings.
  - Form the gravity vector in the body-frame.
  - Form the roll and pitch Euler angles from the gravity unit vector.
- Generates a serial output message<sup>1</sup> consisting of the following:
  - A relative time measurement (both integer and decimal values)
  - Acceleration readings in  $[g]$
  - Rate-sensor readings in  $[\text{°/s}]$
  - Magnetic-field readings in  $[G]$
  - Sensor temperature readings in  $[\text{°C}]$

## 5.4 Vertical-Gyro / Attitude and Heading Reference System Application

The Vertical-Gyro (VG) / Attitude and Heading Reference System (AHRS) application enables the OpenIMU hardware to fuse inertial-sensor information (accelerometers, rate-sensors, and — for the AHRS — magnetometers) to generate an attitude solution. The solution makes use of the high data-rate (DR) rate-sensor output to propagate the attitude forward in time while using the accelerometers and magnetometers as references to correct for estimated rate-bias errors and attitude-errors at a lower DR.

The mathematics behind the algorithm are quite a bit more complicated than the math associated with the Static-Leveler application. The full description is not discussed here, as . However, the complete formulation is provided in the “Ready-to-use Applications” section.

The VG/AHRS example application performs the following functions:

- Sets the default OpenIMU configuration
- Acquires sensor data - acceleration, angular-rate, local magnetic-field, and sensor temperature data
- Executes the VG/AHRS algorithm
- Populates the output data structure
- Generates and sends the following output message to the UART - the output message description is To Be Provided

---

<sup>1</sup> The output message is the same as for the IMU application, but tailored by the Leveler algorithm

## 5.5 Inertial Navigation System Application

The INS APP supports all of the features and operating modes of the VG/AHRS App. In addition it includes the capability of interfacing with an external GPS receiver and associated software running on the processor, allowing computation of navigation information as well as orientation information. The application name, GPS/INS APP, stands for GPS Inertial Navigation System, and it is indicative of the navigation reference functionality that application provides by outputting inertially-aided navigation information (Latitude, Longitude, and Altitude), inertially-aided 3-axis velocity information, as well as heading, roll, and pitch measurements, in addition to digital IMU data.

The mathematics behind the algorithm are more complicated than the math associated with the VG/AHRS application. The full description is not discussed here, as the complete formulation is provided in the “Ready-to-use Applications” section.

The INS example application performs the following functions:

- Sets the default OpenIMU configuration
- Acquires sensor data - acceleration, angular-rate, local magnetic-field, GPS, and sensor temperature data
- Populates the output data structure
- Generates and sends the following output message to the UART - the output message description is To Be Provided

---

## OpenIMU Software Overview

---

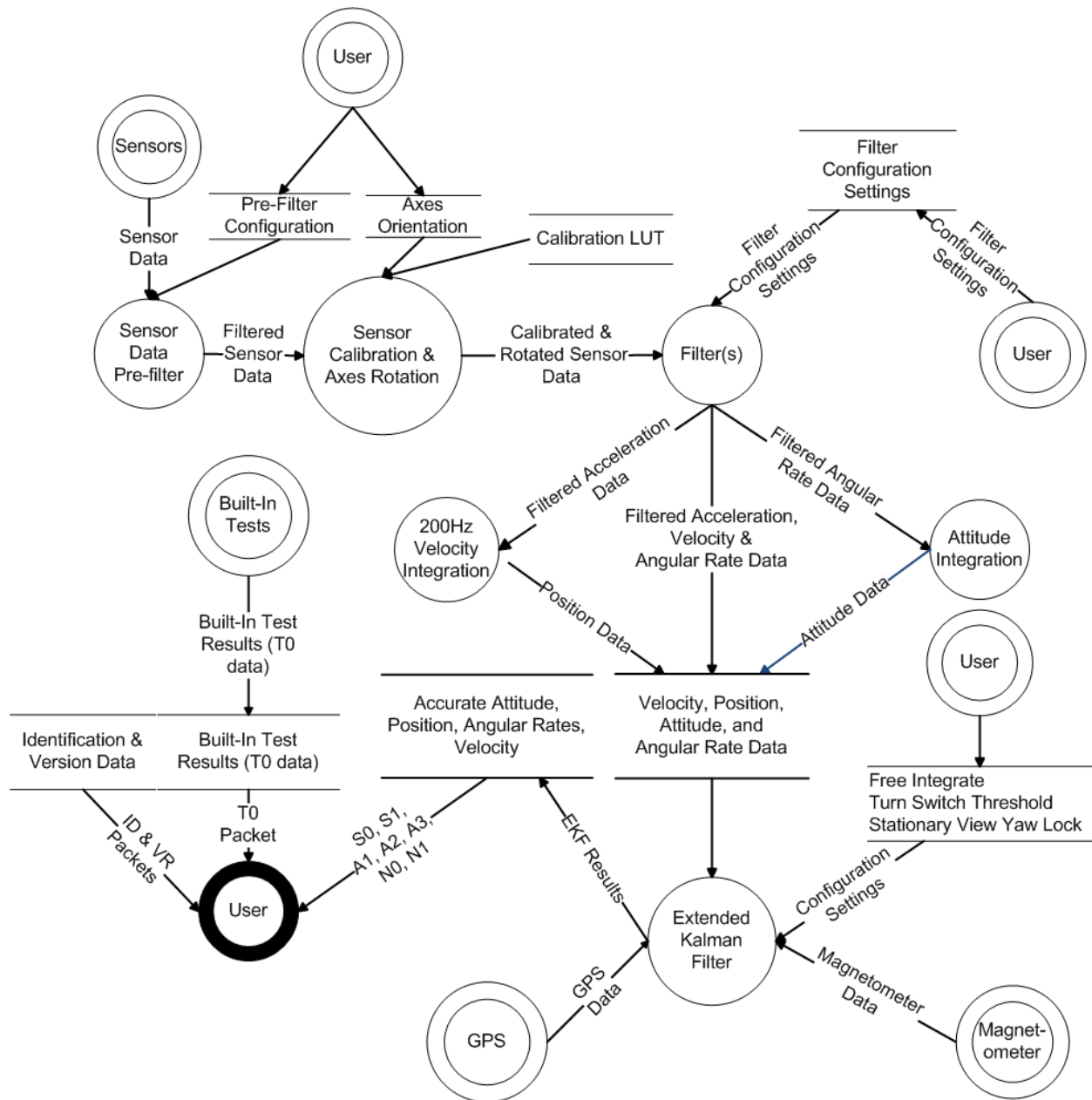
This section reviews more detail on how OpenIMU platform code modules are structured and work together:

- Software Dataflow Diagram
- RTOS
- Sampling and Filtering
- UART Messaging
- SPI Messaging
- Settings
- Tutorial App
- CAN J1939 Messaging

### 6.1 Software DataFlow

**The OpenIMU software data flow is depicted in the following diagram.**

- The double circle icons denote inputs
- The single circle icons denote software components
- The thick single circle icons denote outputs
- the double horizontal line icons denote data stores
- The arrow icons denote data that is sent from one software component, input, or data store to a software component



## 6.2 FreeRTOS

The applications for all OpenIMU300 units use the FreeRTOS Real-Time Operating System ([FreeRTOS Site](#)), while OpenIMU330 units uses a simple real-time scheduler. FreeRTOS is very widely used, as it is feature-rich, has a small footprint, and can be used in commercial application without having to expose intellectual property.

FreeRTOS is licensed under the MIT Open Source License ([FreeRTOS Licence Page](#)).

The FreeRTOS site provides a wealth of informative online documents and PDF books that can be downloaded.

The FreeRTOS source code is supplied, but the user is advised to not change anything in the code.

The many FreeRTOS header files are located in the “*FreeRTOS library/include*” directory. The user is urged to search in that directory when any FreeRTOS related API function prototype, data type, ‘#define’ literal constant, or any other FreeRTOS related item

## 6.3 Sampling and Filtering Modules

To Be Provided

## 6.4 OpenIMU UART Messaging Framework

### 1. General Settings

The serial port settings are: 1 start bit, 8 data bits, no parity bit, 1 stop bit, and no flow control. Standard baud rates supported are: 38400, 57600, 115200, 230400 and 460800.

Common definitions include:

A word is defined to be 2 bytes or 16 bits.

All communications to and from the unit are packets that start with a single word alternating bit preamble 0x5555. This is the ASCII string “UU”.

All communication packets end with a single word CRC (2 bytes). CRCs are calculated on all packet bytes excluding the preamble and CRC itself. Input packets with incorrect CRCs will be ignored.

All multiple byte values except CRC and packet code are transmitted in Little Endian format (Least Significant Byte First).

Each complete communication packet must be transmitted to the OpenIMU300xx inertial system within a 4 second period.

### 2. Number Formats

Number Format Conventions include:

0x as a prefix to hexadecimal values

single quotes (‘’) to delimit ASCII characters

no prefix or delimiters to specify decimal values.

---

**Note:**

- All multiple byte number format are transmitted in little-endian format. E.g., Bytes are transmitted LSB first, followed by lesser significant bytes.

- Bytes in strings are transmitted in left to right string byte order.

The table below defines variable formats:

ID	Type	Size (bytes)	Range
U1	Unsigned Char	1	0 to 255
U2	Unsigned Short	2	0 to 65535
U4	Unsigned Int	4	0 to $2^{32}-1$
U8	Unsigned long long	8	0 to $2^{64}-1$
F	Float IEEE-754	4	$1.18^{-38}$ to $3.4^{38}$
D	Double IEEE-754	8	$2.23^{-308}$ to $1.80^{308}$
I1	Signed Char	1	-128 to +127
I2	Signed Short	2	-32768 to 32767
I4	Signed Int	4	$-2^{31}$ to $2^{31}-1$
I8	Signed long long	8	$-2^{63}$ to $2^{63}-1$
ST	String	N	ASCII

### 3. Packet Structure

Below provided description of OpenIMU framework messages. Messages described the way they occur in serial line. Open IMU framework takes care of wrapping up user payload and calculating CRC.

#### 3.1 Generic Packet Format

All of the Input and Output packets, except the Ping command, conform to the following structure:

0x5555	<2-byte packet code (U2)>	<payload byte-length (U1)>	<variable length payload>	<2-byte CRC (U2)>
--------	---------------------------------	----------------------------------	---------------------------------	----------------------

#### 3.2 Packet Header

The packet header is always the bit pattern 0x5555.

#### 3.3 Packet Code

The packet code is always two bytes long in unsigned short integer format. Most input and output packet types for convenience can be interpreted as a pair of ASCII characters. For example code “aB” will translate to hex value 0x6142”.

NOTE:

1. First character value should be more or equal ‘a’ (0x61)
2. Packet code transmitted in Big Endian format

#### 3.4 Payload Length

The payload length is always a one byte unsigned character with a range of 0-255. The payload length byte is the length (in bytes) of the <variable length payload> portion of the packet ONLY, and does not include the CRC.

#### 3.5 Payload

The payload is of variable length based on the packet type.

### 3.6 16-bit CRC-CCITT

Packets end with a 16-bit CRC-CCITT calculated on the entire packet excluding the 0x5555 header and the CRC field itself. A discussion of the 16-bit CRC-CCITT and sample code for implementing the computation of the CRC is included at the end of this document. This 16-bit CRC standard is maintained by the International Telecommunication Union (ITU). The highlights are:

Width = 16 bits

Polynomial 0x1021

Initial value = 0x1D0F

No XOR performed on the final value.

See Appendix A for sample code that implements the 16-bit CRC algorithm.

### 3.6 NAK Packet

NAK packet sent in response to the unknown or corrupted input message. NAK packet has next format:

0x5555	0x0000	2	code of received packet or 0	<2-byte CRC (U2)>
--------	--------	---	------------------------------	-------------------

## 4. Messaging Overview

Table below summarizes the messages initially introduced in OpenIMU300xx framework. New messages can be easily added (please check chapter “Procedure for adding new message”) Packet codes are assigned mostly using the ASCII mnemonics defined above and are indicated in the summary table below and in the detailed sections for each command. The payload byte-length is often related to other data elements in the packet as defined in the table below. The referenced variables are defined in the detailed sections following. Output messages are sent from the OpenIMU Series inertial system to the user system as a result of user request or a continuous packet output setting. Interactive messages can be sent from the user system to the OpenIMU Series inertial system and will result in an associated Reply Message or NAK message. Note that reply messages typically have the same **<2-byte packet type (U2)>** as the input message that evoked it but with a different payload.

Table 1: Messages Table

ASCII	Code (U2)	Payload Length (U1)	Function	Type
<b>Interactive Messages</b>				
pG	0x7047	0	Ping	Input/Reply Message
uC	0x7543	N (up to 248)	Update Config Command/Response	Input/Reply Message
uP	0x7550	12	Update Parameter Command/Response	Input/Reply Message
uA	0x7541	N (up to 240)	Update All Command/Response	Input/Reply Message
sC	0x7343	0	Save Config Command/Response	Input/Reply Message
rD	0x7244	0	Restore Defaults Command/Response	Input/Reply Message
sC	0x7343	0	Save Config Command/Response	Input/Reply Message
<b>6.4. OpenIMU UART Messaging Framework</b>			Response	<b>73</b>
gC	0x6743	8	Get	Input/Reply



## 5. OpenIMU Interactive Messages

### 5.1 User Ping Command

Ping ('pG' = 0x7047)			
Preamble	Packet Code	Length	Termination
0x5555	0x7047	0	<CRC (U2)>

The user Ping command has no payload. Sending the Ping command will cause the unit to send a Ping response with next format:

Ping ('pG' = 0x7047)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7047	N	Unit Model and Serial Number <S> (string)	<CRC (U2)>

The user Ping response will return null-terminated string, containing unit model name and unit serial number.

### 5.2 Update Config Command

('uC' = 0x7543)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7543	8+8*N	N Parameters	<CRC (U2)>

The Update Config command used to update and apply N consecutive user-defined configuration parameters at a time in unit. Parameter value is 64 bit (8 bytes) and can have arbitrary type.

Update Config Payload Format

Byte Offset	Name	Format	Notes
0	Number of consecutive parameters to update	U4	LSB First
4	Offset of first parameter in unit config structure	U4	LSB First
8	Parameter Value	U8 or I8 or F8 or double or S8 or A8	LSB First
:	:	:	:
8+N*8	Parameter Value	U8 or I8 or F8 or double or S8 or A8	LSB First

Upon reception – each parameter is validated (if desired) and if validation passes parameter gets written into gUserConfiguration structure and also applied to the system on-the-fly(if desired). If value of one parameter is invalid – all parameters ignored. Updated configuration parameters will be active until next unit power cycle or reset.

Update Config command will have next response:

Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7543	4	Error Code (I4)	<CRC (U2)>

Error code can be: (0) – “Success”, (-3) – “Invalid Payload Size”, (-1) – “Invalid parameter number”, (-2) – “Invalid parameter value”

### 5.3 Update Parameter Command

('uP' = 0x7550)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7550	12		<CRC (U2)>

The Update Parameter command used to update and apply single user-defined configuration parameter in unit. Parameter value is 64 bit (8 bytes) and can have arbitrary type.

Update Parameter Payload Format

Byte Offset	Name	Format	Notes
0	Offset of parameter in unit config structure	U4	LSB First
8	Parameter Value	U8 or I8 or F8 or double or S8 or A8	LSB First

Upon reception parameter value is validated (if desired) and if validation passes parameter gets written into gUserConfiguration structure and also applied to the system on-the-fly(if desired). If value of the parameter is invalid – it ignored. Updated configuration parameter will be active until next unit power cycle or reset.

Update Parameter command will have next response:

Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7550	4	Error Code (I4)	<CRC (U2)>

Error code can be: (0) – “Success”, (-3) – “Invalid Payload Size”, (-1) – “Invalid parameter number”, (-2) – “Invalid parameter value”

#### 5.4 Update All Command

('uA' = 0x7541)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7541	8N	N (up to 30) parameters	<CRC (U2)>

The Update All command used to update/apply up to 30 consecutive user-defined configuration parameters at a time in unit, starting from first parameter in user configuration structure. Each parameter has size 8 bytes (64 bit) and can have arbitrary type.

Update All Payload Format

Byte Offset	Name	Format	Notes
0	Parameter Value (first parameter)	U8 or I8 or F8 or double or S8 or A8	LSB First
...	...	...	...
N*8	Parameter Value (last parameter)	U8 or I8 or F8 or double or S8 or A8	LSB First

Upon reception – each parameter is validated (if desired) and if validation passes parameter gets written into gUserConfiguration structure, starting from first parameter (offset 0) and also applied to the system

on-the-fly(if desired). If value of one parameter is invalid – all parameters ignored. First two parameters are ignored. Updated configuration parameters will be active until next unit power cycle or reset.

Update All command will have next response:

Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7541	4	Error Code (I4)	<CRC (U2)>

Error code can be: (0) – “Success”, (-3) – “Invalid Payload Size”, (-1) – “Invalid parameter number”, (-2) – “Invalid parameter value”

### 5.5 Save Config Command

Save Config ('sc' = 0x7343)				
Preamble	Packet Code	Length	Termination	
0x5555	0x7343	0	<CRC (U2)>	

The Save Config command has no payload. Upon reception of “Save Config” command unit will save current gUnitConfiguration structure into EEPROM and updated parameters will be applied to the unit all the times upon startup (untill new changes will be made).

Save Config command will have next response in in case of success:

Preamble	Packet Code	Length	Termination
0x5555	0x7343	0	<CRC (U2)>

**Note:** Save configuration command from serial port works on OpenIMU300ZI. It is not supported by OpenIMU300RI, but one can make permanent changes just by rebuilding the FW with desired default settings.

### 5.5 Restore Defaults\*\*

Restore defaults ('rd' = 0x7244)			
Preamble	Packet Code	Length	Termination
0x5555	0x7244	0	<CRC (U2)>

The Restore defaults command has no payload. Upon reception of “Restore Defaults” command unit will save default configuration structure gDefaultUserConfig into EEPROM and updated parameters will be applied to the unit all the times upon startup (untill new changes will be made).

Restore Defaults command will have next response in case of success:

Preamble	Packet Code	Length	Termination
0x5555	0x7244	0	<CRC (U2)>

### 5.6 Get Config Command

('gC' = 0x6743)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6743	8		<CRC (U2)>

The Get Config command used to retrieve N consecutive user-defined configuration parameters at a time from unit. Parameter value is 64 bit (8 bytes) and can have arbitrary type.

#### Get Config Payload Format

Byte Offset	Name	Format	Notes
0	Number Of consecutive parameters to update	U4	LSB First
4	Offset of first parameter in unit config structure	U4	LSB First

Get Config command will have next response:

('gC' = 0x6743)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6743	8+8*N	N parameters	<CRC (U2)>

Get Config Response Payload Format in case of success:

Byte Offset	Name	Format	Notes
0	Number Of consecutive parameters to update	U4	LSB First
4	Offset of first parameter in unit config structure	U4	LSB First
8	Parameter Value	U8 or I8 or F8 or double or S8 or A8	LSB First
:	:	:	:
8+N*8	Parameter Value	U8 or I8 or F8 or double or S8 or A8	LSB First

Get Config Response Payload Format in case of error:

Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6743	4	Error Code (I4)	<CRC (U2)>

Error code can be: (0) – “Success”, (-3) – “Invalid Payload Size”, (-1) – “Invalid parameter number”, (-2) – “Invalid parameter value”

#### 5.7 Get All Command

('gA' = 0x6741)			
Preamble	Packet Type	Length	Termination
0x5555	0x6741	0	<CRC (U2)>

The Get All command used to retrieve N (up to 30) consecutive user-defined configuration parameters at a time from unit, starting from first parameter in gUserConfiguration structure. Parameter value is 64 bit (8 bytes) and can have arbitrary type.

Get All command will have next response:

Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6741	8*N	N parameters	<CRC (U2)>

Get Config Response Payload Format in case of success:

Byte Offset	Name	Format	Notes
0	Number Of consecutive parameters to update	U4	LSB First
4	Offset of first parameter in unit config structure	U4	LSB First
8	Parameter Value	U8 or I8 or F8 or double or S8 or A8	LSB First
:	:	:	:
8+N*8	Parameter Value	U8 or I8 or F8 or double or S8 or A8	LSB First

Get All Response Payload Format in case of error:

Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6741	4	Error Code (I4)	<CRC (U2)>

Error code can be: (0) – “Success”, (-3) – “Invalid Payload Size”, (-1) – “Invalid parameter number”, (-2) – “Invalid parameter value”

#### 5.8 Get Parameter Command

('gP' = 0x6750)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6750	4		<CRC (U2)>

The Get Parameter command used to retrieve one user-defined configuration parameter from unit gUser-Configuration structure. Parameter value is 64 bit (8 bytes) and can have arbitrary type.

Get Parameter command payload format

Byte Offset	Name	Format	Notes
0	Offset of parameter in unit config structure	U4	LSB First

Get Parameter command will have next response:

('gP' = 0x6750)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6750	12	parameter	<CRC (U2)>

Get Parameter response payload format in case of success:

Byte Offset	Name	Format	Notes
0	Offset of parameter in unit config structure	U4	LSB First
4	Parameter Value	U8 or I8 or F8 or double or S8 or A8	LSB First

Get Parameter response payload format in case of error:

Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6750	4	Error Code (I4)	<CRC (U2)>

Error code can be: (0) – “Success”, (-3) – “Invalid Payload Size”, (-1) – “Invalid parameter number”, (-2) – “Invalid parameter value”

#### 5.9 Get User Version Command

('gV' = 0x6756)			
Preamble	Packet Code	Length	Termination
0x5555	0x6756	0	<CRC (U2)>

The Get Version command has no payload. Sending the Get Version command will cause the unit to send a response with next format:



Preamble	Packet Type	Length	Payload	Termination
0x5555	0x6756	N	User Version String	<CRC (U2)>

The Get Version response will return null-terminated string, user version. User version string defined in the UserMessaging.c file.

## 6. OpenIMU Output messages

Below provided examples of OpenIMU output messages implemented in OpenImu framework. Users can easily add new messages or discard these examples at their discretion. Output messages are to be continuously sent out by unit with preconfigured message rate.

### 6.1 User Test Message

User Test output message has next format:

('zT' = 0x7a54)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7a54	4		<CRC (U2)>

User Test Message payload has next format:

Byte Offset	Name	Format	Notes
0	Counter	U4	LSB First

Counter is simple message counter which will increase by 1 with in every consecutive Test message

### “6.2 User Sensors Data Message”

User Sensors Data message has next format:

('z1' = 0x7a31)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7a31	40		<CRC (U2)>

User Sensors Data Message payload has next format:

Byte Offset	Name	Format	Notes
0	System Timer at the moment of sensors sampling	U4	LSB First
4	Acceleration value for axis X (in G)	F4	LSB First
8	Acceleration value for axis Y (in G)	F4	LSB First
12	Acceleration value for axis Z (in G)	F4	LSB First
16	Rotation speed for axis X (dps)	F4	LSB First
20	Rotation speed for axis Y (dps)	F4	LSB First
24	Rotation speed for axis Z (dps)	F4	LSB First
28	Magnetic field for axis X (G)	F4	LSB First
32	Magnetic field for axis Y (G)	F4	LSB First
36	Magnetic field for axis Z (G)	F4	LSB First

### 6.3 User Arbitrary Data Message

User Arbitrary Data message has next format:

('z2' = 0x7a32)				
Preamble	Packet Type	Length	Payload	Termination
0x5555	0x7a32	27		<CRC (U2)>

User Arbitrary Data Message payload has next format:

Byte Offset	Name	Format	Notes
0	System Timer at the moment of sensors sampling	U4	LSB First
4	Data of type Byte	U1	
5	Data of type short	I2	LSB First
7	Data of type int	I4	LSB First
11	Data of type int64	I8	LSB First
19	Data of type double	D4	LSB First

### 6.4 Raw & Scaled Data Message

Factory Raw Data Packet M (Output Packet) & Scaled Sensor Packet M (Output Packet) has the next format, which is defined in the doc [download](#)

## 7 Steps to create your own interactive or output packet in embedded OpenIMU software framework

User packet processing engine located in the file `UserMessaging.c`.

### 7.1 To create new interactive packet

1. Add new input packet type into the enumerator structure **UserInPacketType** in the file **UserMessaging.h** before **USR\_IN\_MAX**

```
typedef enum {
    USR_IN_NONE          = 0 ,
    USR_IN_PING           ,
    USR_IN_UPDATE_CONFIG ,
    USR_IN_UPDATE_PARAM  ,
    USR_IN_UPDATE_ALL     ,
    USR_IN_SAVE_CONFIG   ,
    USR_IN_RESTORE_DEFAULTS ,
    USR_IN_GET_CONFIG    ,
    USR_IN_GET_PARAM     ,
    USR_IN_GET_ALL       ,
    USR_IN_GET_VERSION   ,
    // add new packet type here, before USR_IN_MAX
    USR_IN_MAX           ,
}UserInPacketType;
```

2. Add new packet type and code into the structure **UserInputPackets** in the file **UserMessaging.c**. Packet code consists of two bytes and can be chosen arbitrary, but first byte SHOULD have value more or equal **0x61**.

```

usr_packet_t userInputPackets[] = {
    {USR_IN_NONE,          {0,0}}, // " "
    {USR_IN_PING,          "pG"},
    {USR_IN_UPDATE_CONFIG, "uC"},
    {USR_IN_UPDATE_PARAM,  "uP"},
    {USR_IN_UPDATE_ALL,    "uA"},
    {USR_IN_SAVE_CONFIG,   "sC"},
    {USR_IN_RESTORE_DEFAULTS, "rD"},
    {USR_IN_GET_CONFIG,    "gC"},
    {USR_IN_GET_PARAM,     "gP"},
    {USR_IN_GET_ALL,       "gA"},
    {USR_IN_GET_VERSION,   "gV"},
    // place new input packet code here, before USR_IN_MAX
    {USR_IN_MAX,           {0xff, 0xff}}, // ""
};

```

3. Add code which handles input packet into the function **HandleUserInputPacket** in the file **UserMessaging.c**. As a part of packet handling fill up desired response payload (starting from address `ptrUcbPacket->payload`) and provide response payload length in the parameter `ptrUcbPacket->payloadLength`. If no response payload required – provide payload length of 0. The packet code in the response will be the same as in the command. If erroneous conditions discovered during packet processing – set valid variable to FALSE so system will respond with NAK packet. Additional diagnostics in arbitrary format can be provided in the response payload (see uP packet example above).

```

case USR_IN_UPDATE_PARAM:
    UpdateUserParam((userParamPayload*)ptrUcbPacket->payload, &
    ptrUcbPacket->payloadLength);
    break;

```

4. Done

## 7.2 To create new output packet

1. Add new output packet type into the enumerator structure **UserOutPacketType** in the file **UserMessaging.h**

```

// User input packet codes, change at will
typedef enum {
    USR_OUT_NONE = 0, // 0
    USR_OUT_TEST,    // 1
    USR_OUT_DATA1 ,  // 2
    USR_OUT_DATA2 ,  // 2
    // add new output packet type here, before USR_OUT_MAX
    USR_OUT_MAX
}UserOutPacketType;

```

2. Add new packet type and code into the structure **UserOutputPackets** in the file **UserMessaging.c**. Packet code can be chosen arbitrary, but first byte SHOULD have value more or equal 0x61 and the packet code should be unique among input and output packets.

```

// packet codes here should be unique -
// should not overlap codes for input packets and system packets
// First byte of Packet code should have value >= 0x61
usr_packet_t userOutputPackets[] = {
    // Packet Type          Packet Code
    {USR_OUT_NONE,          {0x00, 0x00}},
    {USR_OUT_TEST,          "zT"},

```

(continues on next page)

(continued from previous page)

```
{USR_OUT_DATA1,          "z1"},
{USR_OUT_DATA2,          "z2"},
// place new type and code here
{USR_OUT_MAX,            {0xff, 0xff}}, // ""
};
```

3. Add code which handles input packet into the function `HandleUserOutputPacket` in the file `UserMessaging.c`. Fill up desired packet payload (starting from address payload) and provide response payload length in the parameter `payloadLen`. If no response payload required – provide payload length of 0.

```
case USR_OUT_DATA1:
{
    int n = 0;
    double accels[3];
    double mags[3];
    double rates[3];
    data1_payload_t *pld = (data1_payload_t *)payload;

    pld->timer = platformGetDacqTime();
    GetAccelData_mPerSecSq(accels);
    for (int i = 0; i < 3; i++, n++){
        pld->sensorsData[n] = (float)accels[i];
    }
    GetRateData_degPerSec(rates);
    for (int i = 0; i < 3; i++, n++){
        pld->sensorsData[n] = (float)rates[i];
    }
    GetMagData_G(mags);
    for (int i = 0; i < 3; i++, n++){
        pld->sensorsData[n] = (float)mags[i];
    }
    *payloadLen = sizeof(data1_payload_t);
}
```

4. To activate output of the packet use function `SetUserPacketType` in file `UserMessaging.c` and provide desired packet type as a parameter. Or provide output packet type and packet rate in default user configuration in file `UserConfiguration.c`. Output of specific packet can also be changed “on-the-fly” by sending to unit command “uP” with parameter number 3 and desired parameter value. Output packet rate can be changed “on-the-fly” by sending to unit command “uP” with parameter number 4 and desired parameter value.

```
// Default user configuration structure
// Saved into EEPROM of first startup after reloading the code
// or as a result of processing "rD" command
// Do Not remove - just add extra parameters if needed
// Change default settings if desired
const UserConfigurationStruct gDefaultUserConfig = {
    .dataCRC          = 0,
    .dataSize         = sizeof(UserConfigurationStruct),
    .userUartBaudRate = 115200,
    .userPacketType   = "z1",
    .userPacketRate   = 50,
    .lpfAccelFilterFreq = 50,
    .lpfRateFilterFreq = 50,
    .orientation      = "+X+Y+Z"
// add default parameter values here, if desired
```

(continues on next page)

(continued from previous page)

```
} ;
```

5. Done

## 6.5 OpenIMU SPI Messaging Framework

### 1. Introduction

OpenIMU supports a SPI interface for data communications as a one of the choices. To enforce SPI interface mode 'Data Ready' signal needs to be forced HIGH of left unconnected on system startup. OpenIMU SPI interface signals described [here](#).

OpenIMU operates as a slave device.

### 2. OpenIMU SPI communication model

OpenIMU has 128 8-bit registers accessible via SPI interface for reading and writing. The usage of these registers is completely user-defined in time of FW development. Access to the few registers is implemented in the examples as a reference:

Table 1. SPI registers used in the examples

Register Number	Access Type OpenIMU300ZI	Access Type OpenIMU330BI	Function	Notes
4,5 (0x04, 0x5)	r	r	X-Rate	MSB in reg.4 (Note 1)
6,7 (0x06, 0x7)	r	r	Y-Rate	MSB in reg.6 (Note 1)
8,9 (0x08, 0x9)	r	r	Z-Rate	MSB in reg.8 (Note 1)
10,11 (0x0A, 0xB)	r	r	X-Accel	MSB in reg.10 (Note 2)
12,13 (0x0C, 0xD)	r	r	Y-Accel	MSB in reg.12 (Note 2)
14,15 (0x0E, 0xF)	r	r	Z-Accel	MSB in reg.14 (Note 2)
16,17 (0x10, 0x11)	r	N/A	X-MAG	MSB in reg.16 (Note 3)
18,19 (0x12, 0x13)	r	N/A	Y-MAG	MSB in reg.18 (Note 3)
20,21 (0x14, 0x15)	r	N/A	Z-MAG	MSB in reg.20 (Note 3)
22,23 (0x16, 0x17)	r	r	Board-Temp	MSB in reg.22 (Note 4)
24,25 (0x18, 0x19)	r	r	Sensor-Temp	MSB in reg.24 (Note 4)
50 (0x32 )	r	N/A	Mag Scale Factor	(TBD)
55 (0x37 )	r/w	r/w	Drdy Rate	See p.8
56 (0x38 )	r/w	r/w	Accel LPF	See p.7

Continued on next page

Table 2 – continued from previous page

61 (0x3D )	r	r	Burst Read VG	VG Application See p.4
62 (0x3E )	r	r	Burst Read	See p.4
61 (0x3F )	r	N/A	Burst Read MAG	IMU Application See p.4
70 (0x46 )	r	r	Accel Scale Factor	see p.13
71 (0x47 )	r	r	Rate Scale Factor	see p.14
72 (0x48,0x49)	r	N/A	MAGX Hard Iron	MSB in reg.72 (TBD)
74 (0x4A,0x4B)	r	N/A	MAGY Hard Iron	MSB in reg.74 (TBD)
76 (0x4C,0x4D)	r	N/A	MAG SF Soft Iron	MSB in reg.76 (TBD)
78 (0x4E,0x4F)	r	N/A	MAG Angle Soft	MSB in reg.78 (TBD)
80 (0x50)	r/w	N/A	MAG Align com- mand	(TBD)
81 (0x51)	r	N/A	MAG Align sta- tus	(TBD)
82,83,84,88,89 (0x52 0x53,0x54,0x58,0x59)	r	r	Unit serial number	BCD format
86, 87 (0x56, 0x57)	r	r	Product ID	BCD format 3000 - Open- IMU300 3300 - Open- IMU330
90,91 (0x5A, 0x5B)	r	r	Master status	see.p 10
92,93 (0x5C, 0x5D)	r	r	HW status	see.p 11
94,95 (0x5E, 0x5F)	r	r	SW status	see.p 12
112 (0x70)	r	r/w	Accel Range	see.p 13
113 (0x71)	r	r/w	Rate Range	see.p 14
116,117 (0x74,0x75)	r/w	r/w	Unit Orientation	MSB in reg.78 See p.6
118 (0x76)	r/w	r/w	Save Configura- tion	See p.9
120 (0x78)	r/w	r/w	Rate LPF	See p.7

Continued on next page

Table 2 – continued from previous page

126 (0x7E)	r	r	HW Version	
127 (0x7F)	r	r	SW Version	

Note 1: Rate sensors scale is 64 LSB/dps. See p.13

Note 2: Accelerometer sensors scale is 4000 LSB/G. See p.14

Note 3: Magnetometer sensors scale is 16354 LSB/Gauss.

Note 4: Temperature sensors data conversion:

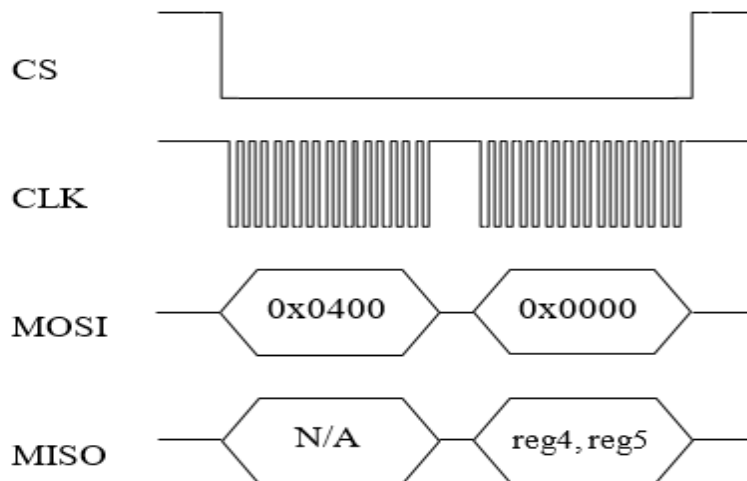
$$\text{Temperature (deg C)} = \text{Temp\_Register\_Value} * 0.073111172849435 + 31.0$$

### 3. OpenIMU SPI Register Read Methodology

SPI master initiates a register read (for example register 0x04) by clocking in the address followed by 0x00, i.e. 0x0400, via MOSI. This combination is referred to as a read-command. It is followed by 16 zero-bits to complete the SPI data-transfer cycle. As the master transmits the read command over MOSI, the OpenIMU transmits information back over MISO.

In this transmission, the first data-word sent by the OpenIMU (as the read-command is sent) consists of 16-bits of non-applicable data. The subsequent 16-bit message contains information stored inside two consecutive registers (in this case registers 4 (MSB) and 5(LSB)).

**Figure 1 illustrates register read over SPI interface:**



### 4. OpenIMU SPI Block Mode Read Methodology

User can implement reading blocks of data with arbitrary length and information. Specific dedicated register address will indicate request specific block of data.

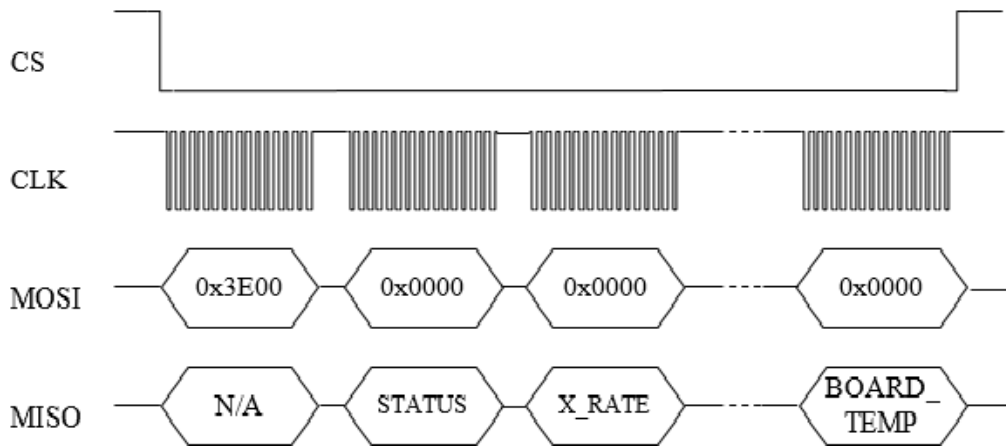
For example, register address 0x3e (62) indicates request for reading data block containing current data from unit sensors. Next table lists corresponding parameters:



Table 2. Block mode message format

Parameter Number	Size (bytes)	Description
Status	2	Unit Status (see.p 10)
X_Rate	2	Rate Sensor output (X) (64 LSB/deg/s)
Y_Rate	2	Rate Sensor output (Y) (64 LSB/deg/s)
Z_Rate	2	Rate Sensor output (Z) (64 LSB/deg/s)
X_Accel	2	Accel Sensor output(X) (4000LSB/G)
Y_Accel	2	Accel Sensor output(Y) (4000 LSB/G)
Z_Accel	2	Accel Sensor output(Z) (4000 LSB/G)
Temp	2	Unit Temperature

Read of data block begins when the master requests a read from specific register address (i.e. 0x3E). **Figure 2 illustrates the read sequence:**



Note: Number of SPI clock pulses should be exactly equal to the length of predefined data packet (in this case – 144 (16 for address 128 for data)) otherwise interface may go out of sync.

For VG\_AHRS/INS application examples next block message supported (register 0x3D):

Table 3. Extended VG block mode message format

Parameter Number	Size (bytes)	Description
Status	2	Unit Status (see p.10)
X_Rate	2	Rate Sensor output (X) (64 LSB/deg/s)
Y_Rate	2	Rate Sensor output (Y) (64 LSB/deg/s)
Z_Rate	2	Rate Sensor output (Z) (64 LSB/deg/s)
X_Accel	2	Accel Sensor output(X) (4000LSB/G)
Y_Accel	2	Accel Sensor output(Y) (4000 LSB/G)
Z_Accel	2	Accel Sensor output(Z) (4000 LSB/G)
Temp	2	Unit Temperature
Roll	2	Unit Roll Angle (65536/(2*PI))LSB/RAD
Pitch	2	Unit Pitch Angle (65536/(2*PI))LSB/RAD
Yaw	2	Unit Yaw angle (65536/(2*PI))LSB/RAD

For units with built-in magnetometer (OpenIMU330ZI) in some application examples next block message supported (register 0x3F):

Table 4. Extended MAG block mode message format

Parameter Number	Size (bytes)	Description
Status	2	Unit Status see p.10
X_Rate	2	Rate Sensor output (X) (64 LSB/deg/s)
Y_Rate	2	Rate Sensor output (Y) (64 LSB/deg/s)
Z_Rate	2	Rate Sensor output (Z) (64 LSB/deg/s)
X_Accel	2	Accel Sensor output(X) (4000LSB/G)
Y_Accel	2	Accel Sensor output(Y) (4000 LSB/G)
Z_Accel	2	Accel Sensor output(Z) (4000 LSB/G)
Temp	2	Unit Temperature
MAG_X	2	Mag sensor output (X) (16384 bits/Gauss)
MAG_Y	2	Mag sensor output (Y) (16384 bits/Gauss)
MAG_Z	2	Mag sensor output (Z) (16384 bits/Gauss)

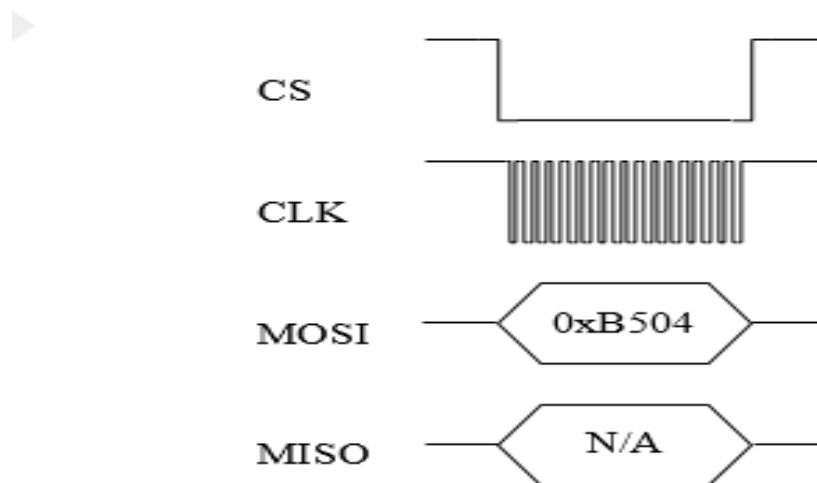
## 5. OpenIMU SPI Register Write Methodology

The SPI master device can perform write into any register. The unit reaction on write operation is completely defined by the user. By default, corresponding data written without any reaction from unit. Written data can be read back. Unlike reads, writes are performed one byte at a time.

The following example highlights how write-commands are formed:

- Select the write address of the desired register, for example 0x35
- Change the most-significant bit of the register address to 1 (the write-bit), e.g. 0x35 becomes 0xB5
- Create the write command by appending the write-bit/address combination with the value to be written to the register (for example 0x04) - 0xB504

Figure 3 illustrates the register write over SPI:



## 6. OpenIMU Orientation programming

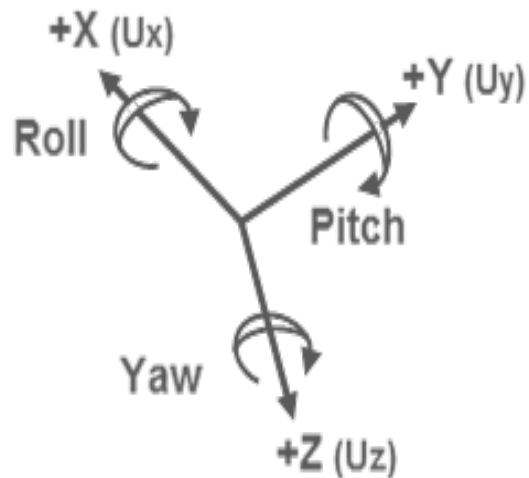
OpenIMU Orientation can be changed dynamically by writing corresponding values into the SPI registers 0x74 (MSB) and 0x75 (LSB). Data into register 0x74 should be written first. There are 24 possible orientation configurations (see

below). Setting/Writing the field to anything else has no effect.

Table 5. OpenIMU Orientation field values

Registers 0x74/0x75	X	Y	Z
0x0000	+Ux	+Uy	+Uz
0x0009	-Ux	-Uy	+Uz
0x0023	-Uy	+Ux	+Uz
0x002A	+Uy	-Ux	+Uz
0x0048	+Ux	-Uy	-Uz
0x0062	+Uy	+Ux	-Uz
0x006B	-Uy	-Ux	-Uz
0x0085	-Uz	+Uy	+Ux
0x008C	+Uz	-Uy	+Ux
0x0092	+Uy	+Uz	+Ux
0x009B	-Uy	-Uz	+Ux
0x0041	-Ux	+Uy	-Uz
0x00C4	+Uz	+Uy	-Ux
0x00CD	-Uz	-Uy	-Ux
0x00D3	-Uy	+Uz	-Ux
0x00DA	+Uy	-Uz	-Ux
0x0111	-Ux	+Uz	+Uy
0x0118	+Ux	-Uz	+Uy
0x0124	+Uz	+Ux	+Uy
0x012D	-Uz	-Ux	+Uy
0x0150	+Ux	+Uz	-Uy
0x0159	-Ux	-Uz	-Uy
0x0165	-Uz	+Ux	-Uy
0x016C	+Uz	-Ux	-Uy

The default factory axis setting for the OpenIMU300ZI for SPI interface is (-Uy, -Ux, -Uz) which defines the connector pointing in the +Z direction and the +X direction going from the connector through the serial number label at the end of the unit. The user axis set (X, Y, Z) as defined by this field setting is depicted in figure below: **Figure 4 illustrates unit orientation:**



## 7. OpenIMU Digital Low Pass Filter selection

OpenIMU low pass filters can be changed dynamically for accelerometers and rate sensors writing corresponding values into the SPI registers 0x38 (for accelerometers) and 0x78 (for rate sensors). There are 11 possible low pass filter options (see below). Setting/Writing the field to anything else has no effect.

Table 6. OpenIMU Digital filter choices

Value Hex (dec)	Cutoff Frequency	Filter Type
0x00 (0)	N/A	Unfiltered
0x03 (3)	40 Hz	Bartlett
0x04 (4)	20 Hz	Bartlett
0x05 (5)	10 Hz	Bartlett
0x06 (6)	5 Hz	Bartlett
0x30 (48)	50 Hz	Butterworth
0x40 (64)	20 Hz	Butterworth
0x50 (80)	10 Hz	Butterworth
0x60 (96)	5 Hz	Butterworth

## 8. OpenIMU DATA READY signal rate

OpenIMU DATA READY signal rate can be changed dynamically by writing corresponding values into the SPI register 0x37. There are 10 possible options (see below). Setting/Writing the field to anything else has no effect.

Table 7. OpenIMU SPI ODR Rate choices

Value Hex (dec)	Data Ready signal rate (Hz)
0x00 (0)	0
0x01 (1)	200 Hz (default)
0x02 (2)	100 Hz
0x03 (3)	50 Hz
0x04 (4)	25 Hz
0x05 (5)	20 Hz
0x06 (6)	10 Hz
0x07 (7)	5 Hz
0x08 (8)	4 Hz
0x09 (9)	2 Hz

## 9. Saving unit configuration

Some configuration parameters can be saved in EEPROM and become active upon next unit restart (reset or power cycle). To save all parameters value 0 or 0xFF needs to be written to the register 0x76. It's possible to save only specific parameter writing corresponding register address into register 0x76. Valid addresses are: 0x37, 0x38, 0x70, 0x71, 0x74, 0x75, 0x78.

## 10. Master Status Register

Master status register reflects current status of the unit. Next status indication bits are available:

Table 8. Master Status Register

<i>Bit</i>	<i>Status</i>
0	Master Fail (1 - error)
1	HW Error (1 - error)
2	Reserved
3	SW Error (1 - error)
4 - 11	Reserved
12	Sensor Status (1 - error)
13 - 15	Reserved

## 11. HW Status Register

HW status register reflects current status of the unit. Next status indication bits are available:

Table 9. HW Status Status Register

<i>Bit</i>	<i>Status</i>
0 - 1	Reserved
2	Sensor Error (1 - error)
3	Mag Error (1 - error)
4	Accel Error (1 - error)
5	Gyro Error (1 - error)
6 - 15	Reserved

## 12. SW Status Register

SW status register reflects current status of the unit. Next status indication bits are available:

Table 10. SW Status Status Register

<i>Bit</i>	<i>Status</i>
0	Algorithm Error (1 - error)
1	Data Error (1 - error)
2	Cal CRC Error (1 - error)
3 - 15	Reserved

## 14. Accelerometer sensors scale factors and range

Next accelerometer scale factors and ranges are applicable:

Table 11. Accelerometer sensors scale factors and ranges

Unit	Range & scale factor	Value in register 0x70	Value in register 0x46
OpenIMU300ZI	8G , 4000 LSB/G	8 (r)	4 (r)
OpenIMU330BI	8G , 4000 LSB/G	8 (r/w)	4 (r)
OpenIMU330BI	16G , 2000 LSB/G	16 (r/w)	2 (r)

## 15. Rate sensors scale factors and range

Next rate sensors scale factors and ranges are applicable:

Table 11. Rate sensors scale factors and ranges

Unit	Range & scale factor	Value in register 0x71	Value in register 0x47
OpenIMU300ZI	500 dps, 64 LSB/dps	8 (r)	64 (r)
OpenIMU330BI	500 dps, 64 LSB/dps	8 (r/w)	64 (r)
OpenIMU330BI	1000 dps, 32 LSB/dps	16 (r/w)	32 (r)
OpenIMU330BI	2000 dps, 16 LSB/dps	32 (r/w)	16 (r)

## 16. Suggested Operation

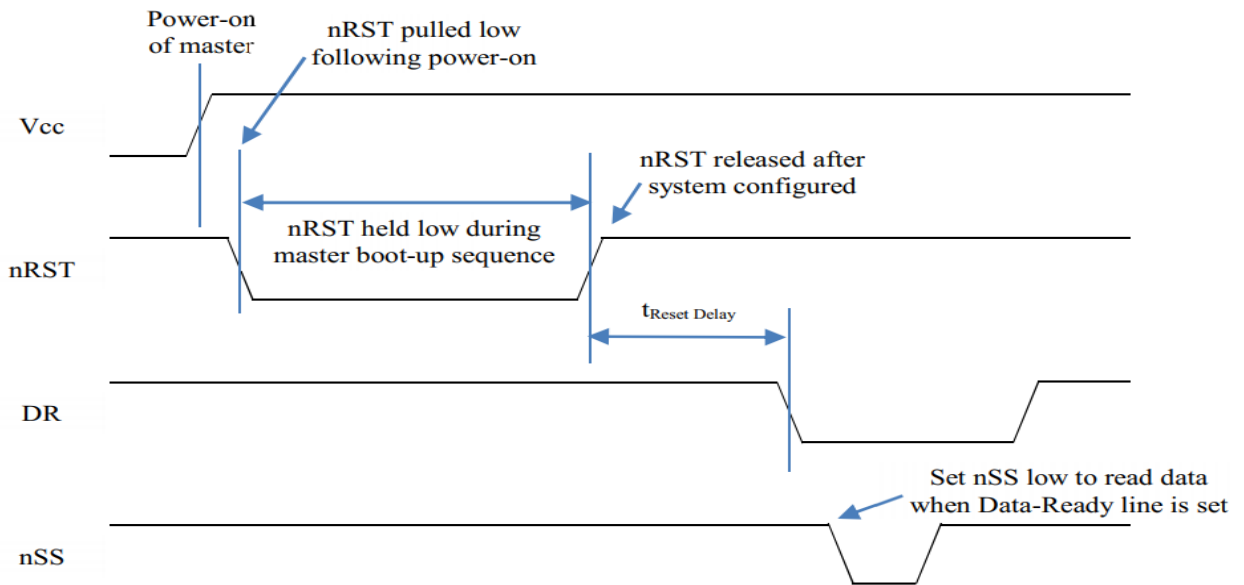
The following operational procedure and timing specifications should be adhered to while communicating with the OpenIMU300/OpenIMU330 via SPI interface to ensure proper system operation. These points are further highlighted later in this section.

### Startup Timing

The following timing applies at system startup (Figure 4):

- During system setup, the OpenIMU should be held in reset (nRST line held low) until the SPI master is configured and the system is ready to begin communications with the OpenIMU
- After releasing the reset line, the OpenIMU requires about 250-500 msec ( $t_{\text{System Delay}}$ ) before the system is ready for use
- Data best to be read from the OpenIMU right after falling edge of DATA READY signal. But at if readings are not synced to DATA READY signal - the latest available data sample will be provided.

Figure 5 illustrates OpenIMU startup timings:



**Figure 5. Startup Timing**

### SPI Timings

The timing requirements for the SPI interface are listed in Table 12 and illustrated in Figure and Figure. In addition, the following operational constraints apply to the SPI communications:

- The unit operates with CPOL = 1 (polarity) and CPHA = 1 (phase)
- Data is transmitted 16-bits words, Most Significant Bit (MSB) first

Table 12. SPI Timing Requirements

Parameter	Description	Value for OpenIMU300ZI	Value for OpenIMU330BI	Units
fCL	SPI clock frequency	1	1.2	MHz
tDELAY	Time between successive clock cycles	9 (min)	16	uSec
tSU,NSS	nSS setup time prior to clocking data	133	133	nSec
th,NS	nSS hold time following clock signal	100	100	nSec
tV,MISO	Time after falling edge of previous clock-edge that MISO databit is invalid	25	25	nSec
tSU,MOSI	Data input setup time prior to rising edge of clock	25	25	nSec
th,MOSI	Data input hold time following rising edge of clock	8	8	nSec

Figure 6 illustrates OpenIMU SPI bus timings:

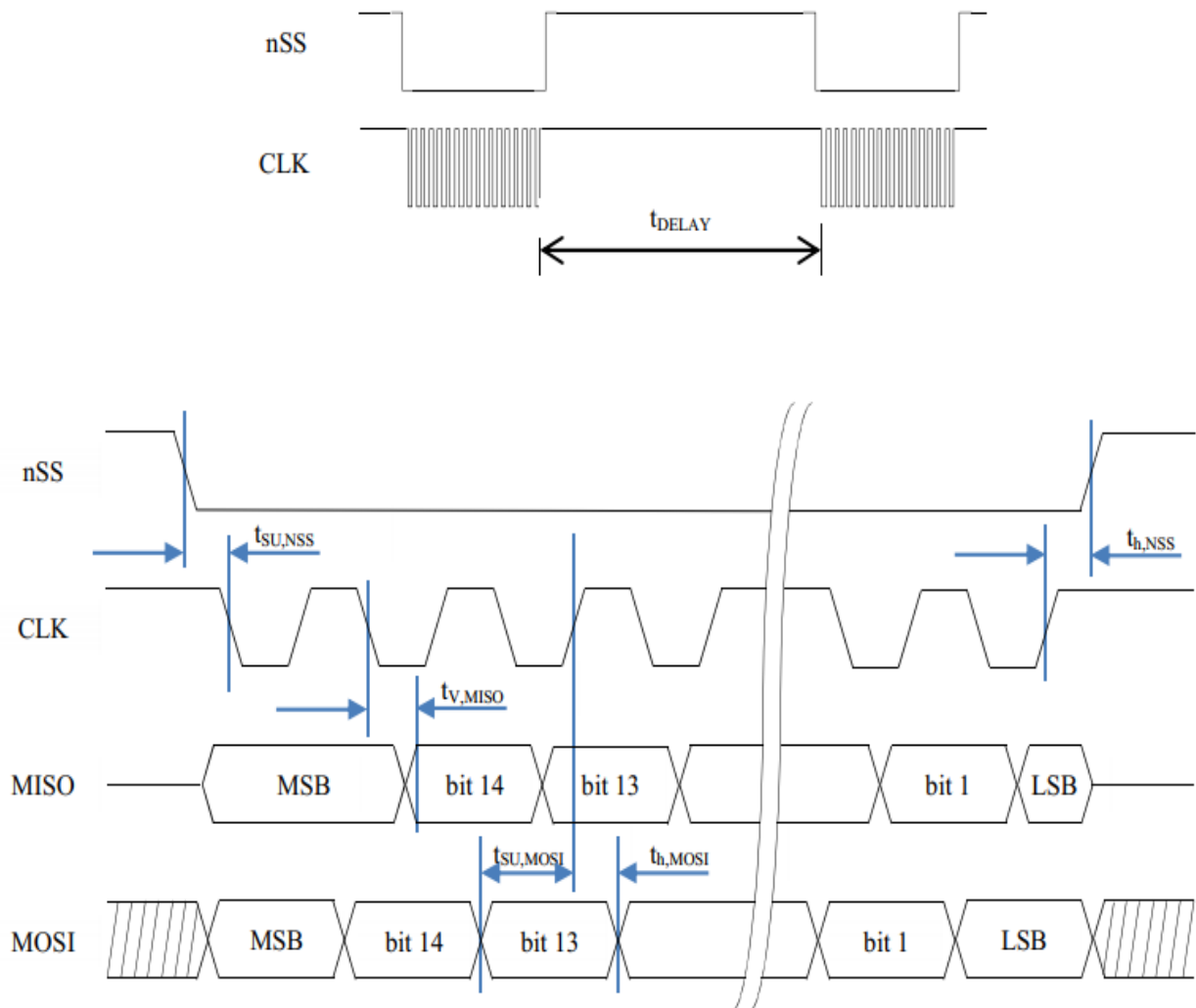


Figure 6. SPI Timing Diagram

## 6.6 Settings Modules

### Configuration parameters in EEPROM

OpenIMU software framework provides possibility for user to store arbitrary configuration parameters in nonvolatile EEPROM. These parameters will be validated and applied to system upon reset or power-up. Parameters which passed validation will override default factory settings. User EEPROM has size 16KB. Each parameter in user EEPROM has size 8 bytes (64-bit word), so user EEPROM can contain up to 2K parameters. If desired one can use few consecutive parameters to store arbitrary value or data structure. One parameter is good for a value of double or long long type. Also it can be considered as 8 bytes of arbitrary data (string or array). There are few pre-allocated recommended parameters which can be useful while working with OpenIMU software framework. Initial definition of parameters structure located in file UserConfiguration.h. New arbitrary parameters are welcome.



```

/// User defined configuration structure
///Please notice, that parameters are 64 bit to accommodate double types as
↪well as string or byte array types

```

```

typedef struct { uint64_t dataCRC; /// CRC of user configuration structure CRC-16 uint64_t dataSize;
    /// Size of the user configuration structure

    int64_t userUartBaudRate; /// baud rate of user UART, bps. /// valid options are: /// 4800 ///
    9600 /// 19200 /// 38400 /// 57600 /// 115200 /// 230400 /// 460800

    uint8_t userPacketType[8]; /// User packet to be continuously sent by unit /// Packet types de-
    fined in structure UserOutPacketType /// in file UserMessaging.h

    int64_t userPacketRate; /// Packet rate for continuously output packet, Hz. /// Valid settings
    are: 0 ,2, 5, 10, 20, 25, 50, 100, 200

    int64_t lpfAccelFilterFreq; /// built-in lpf filter cutoff frequency selection for accelerometers int64_t
    lpfRateFilterFreq; /// built-in lpf filter cutoff frequency selection for rate sensors

    /// Options are: /// 0 - Filter turned off /// 50 - Butterworth LPF 50HZ /// 20 - Butterworth
    LPF 20HZ /// 10 - Butterworth LPF 10HZ /// 05 - Butterworth LPF 5HZ /// 02 - Butterworth
    LPF 2HZ /// 25 - Butterworth LPF 25HZ /// 40 - Butterworth LPF 40HZ

    uint8_t orientation[8]; /// unit orientation as string /// "SFSRSD" /// Where S is sign (+ or -) /// F -
    forward axis (X or Y or Z) /// R - right axis (X or Y or Z) /// D - down axis (X or Y or Z) /// For
    example "+X+Y+Z"

    //*****
    // here is the border between arbitrary parameters and platform configuration parameters
    //*****

    // place new arbitrary configuration parameters here // parameter size should even to 8 bytes // Add
    parameter offset in UserConfigParamOffset structure if validation or // special processing required

} UserConfigurationStruct;

```

### Default configuration

Default system parameters reside in the **gDefaultUserConfig** structure in file **UserConfiguration.c**. They are becoming active each time new application image is loaded to the unit or upon reception of the "rD" command.

### Mapping different values into 64-bit parameter

Below provided recommended mapping of the values of different types into 64-bit parameter. The mapping though can be arbitrary and in that case should be processed accordingly.

1. Mapping of 4-byte integer into 64-bit parameter (value is in Little Endian format)

0	1	2	3	4	5	6	7
LSB			MSB	0	0	0	0

2. Mapping of 2-byte integer into 64-bit parameter (value is in Little Endian format)

0	1	2	3	4	5	6	7
LSB	MSB	0	0	0	0	0	0

3. Mapping of 4-byte floating point value into 64-bit parameter (value is in Little Endian format)

0	1	2	3	4	5	6	7
LSB			MSB	0	0	0	0

4. Mapping of 8-byte double value into 64-bit parameter (value is in Little Endian format)

0	1	2	3	4	5	6	7
LSB							MSB

5. Mapping byte array or string into 64-bit parameter

Byte (character) indexes match offset in the 64-bit parameter

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

#### *Adding new parameter*

One can arbitrary add new configuration parameters. The steps are:

1. Add required parameter into the **UserConfigurationStruct** in the file **UserConfiguration.h** after system parameters “border” (see above).
2. Add new configuration parameter enumerator into UserConfigParamOffset in the file UserConfiguration.h after USER\_LAST\_SYSTEM\_PARAM.
3. Add default value of new parameter into structure gDefaultUserConfig in file UserConfiguration.c (if desired)
4. Add validation of new parameter into function UpdateUserParameter (if desired) or explicitly use parameter at your discretion

#### **Changing configuration parameters**

Configuration parameters can be changed any time by sending specific commands (messages) to the unit (“uP” “uA” “uC”). Upon reception of corresponding message parameters are validated (if desired), placed into gUserConfiguration structure and applied to the unit (if desired). See section Messaging Modules for details. Updated parameters will last until unit reset or power cycle.

#### **Retrieving configuration parameters.**

Configuration parameters can be read from unit any time by sending commands “gC” “gP” or “gA” (see messaging-modules).

#### **Saving configuration parameters**

If desired, updated parameters can be saved into EEPROM and will be permanently active until changed. It can be achieved by sending “sC” command to the unit. Upon reception of this command gUserConfiguration structure saved into EEPROM.

#### **Restoring default configuration**

If desired, default configuration can be restored and saved into EEPROM. It can be achieved by sending command “rD” to the unit.

## **6.7 Tutorial APP**

A simple static tilt sensor demo is provided here to show how to add your own algorithm and output algorithm results.

OpenIMU provides a user-friendly interface to add your own algorithms. To do that, the user need to get sensor data, run the algorithm and output algorithm results. All interfaces related to these operations are handled in `src/dataProcessingAndPresentation.c`. And all user codes implementing the algorithms and results packaging are located in `src/user/` directory.

### 6.7.1 Get algorithm input

The platform provides APIs to access all available sensor data, as shown in the following table.

Sensors	Get sensor data API
Accelerometer	<code>void GetAccelsData(double *data)</code>
Gyroscope	<code>void GetRatesData(double *data)</code>
Magnetometer	<code>void GetMagsData(double *data)</code>
GPS	<code>void GetGPSData(gpsDataStruct_t *data)</code>
Accelerometer temperature	<code>void GetAccelsTempData(double *temps)</code>
Gyroscope temperature	<code>void GetRatesTempData(double *temps)</code>
Board temperature	<code>void GetBoardTempData(double *temp)</code>

Usually, the accelerometer and gyroscope data are already temperature-calibrated.

### 6.7.2 Run the algorithm

A user defined algorithm should provide its main procedure as:

```
void *RunUserNavAlgorithm(double *accels, double *rates, ....., int dacqRate)
```

where **accels** and **rates** are pointers to corresponding sensor measurements, and `dacqRate` is the sensor sampling rate.

This procedure is implemented in `src/user/userAlgorithm.c` as follows:

```
void *RunUserNavAlgorithm(double *accels, double *rates, double *mags,
                          gpsDataStruct_t *gps, int dacqRate)
{
    //-----get accel data-----
    float a[3]; // accel of this step
    a[0] = accels[0];
    a[1] = accels[1];
    a[2] = accels[2];

    //-----calculate euler angles-----
    results[2] = a[0];
    results[3] = a[1];
    results[4] = a[2];
    float accel_norm = sqrt(a[0]*a[0] + a[1]*a[1] + a[2]*a[2]);
    a[0] /= accel_norm;
    a[1] /= accel_norm;
    a[2] /= accel_norm;
    results[0] = asin(a[0]) * R2D;
    results[1] = atan2(-a[1], -a[2]) * R2D;

    //-----return results-----
    return &results;
}
```

It just gets the accelerometer measurement, normalizes it, calculates pitch and roll angles, and returns the results. I keep all the input parameters here. Indeed, I need only **accels**. The user could remove unused parameters in your algorithm.

**results** is a global variable declared as

```
// algorithm results, [pitch roll ax ay az], in units of deg and g
static float results[5];
```

and **R2D** is a macro converting radian to degree:

```
#define R2D 57.2957795130823
```

User may also need to implement an algorithm initialization procedure. It is not necessary in this demo, but will be shown here.

```
void InitUserAlgorithm()
{
    // place additional required initialization here
    // initialize sample rate and period
    results[0] = 0.0;
    results[1] = 0.0;
}
```

Now, a simple user-fined algorithm is done. The framework will automatically call **InitUserAlgorithm** at the initialization stage, and periodically call **RunUserNavAlgorithm** to run the user-defined algorithm and get results.

### 6.7.3 Output results via debug UART

This section shows how to use the debug UART (default baud rate is 38400) on the EVB to output algorithm results. The user could also output other information the user are interested in.

To use the debug UART, the user needs to include **debug.h**. For example, I want to output algorithm results after the algorithm is called in **dataProcessingAndPresentation.c**.

- include the header file in **dataProcessingAndPresentation.c**.

```
#include "debug.h"
```

- output algorithm results. The results are converted to plain text and then transmitted via the debug UART. The user can also choose to encode the results per user requirements.

```
// Output results via debug UART. Downsampled by osr due to limited UART bandwidth
static int out_cntr = 0;
int osr = 8;
out_cntr++;
if(out_cntr==osr)
{
    out_cntr = 0;
    // generate output string from results
    float *tlt = (float*)results;
    char buffer[128];
    sprintf(buffer,
            "pitch:%.3f\troll:%.3f\tax:%.3f\tay:%.3f\taz:%.3f\n",
            tlt[0], tlt[1], tlt[2], tlt[3], tlt[4]);
    // output to debug UART
    DebugPrintString(buffer);
}
```

Compile the project, upload the firmware, and the user can get result via debug UART.

## 6.7.4 Implementing user-defined packets via UART

The debug UART is mainly intended for debug usage. The user may want to output algorithm results via the interface UART (default baud rate is 57600) on the EVB. OpenIMU provides an easy-to-use framework for the user to define your own packets. User-defined packets are declared and implemented in **UserMessaging.h** and **UserMessaging.c**.

- Add your packet code in **UserMessaging.h**.

I added a **USR\_OUT\_TLT** packet as an example.

```
// User input packet codes, change at will
typedef enum {
    USR_OUT_NONE = 0, // 0
    USR_OUT_TEST, // 1
    USR_OUT_DATA1, // 2
    USR_OUT_TLT, // 3
    // place output packet definitions here
    USR_OUT_MAX
}UserOutPacketType;
```

- Add encoding procedure in **UserMessaging.c**.

User defined packets are encoded by this procedure:

```
BOOL HandleUserOutputPacket(uint8_t *payload, uint8_t *payloadLen)
```

After I added my encoding codes, this procedure is as follows.

```
BOOL HandleUserOutputPacket(uint8_t *payload, uint8_t *payloadLen)
{
    static uint32_t _testVal = 0;
    BOOL ret = TRUE;

    switch (_outputPacketType) {
    case USR_OUT_TEST:
        { uint32_t *testParam = (uint32_t*)(payload);
          *payloadLen = USR_OUT_TEST_PAYLOAD_LEN;
          *testParam = _testVal++;
        }
        break;
    case USR_OUT_DATA1:
        { int n = 0;
          double accels[3];
          double mags[3];
          double rates[3];
          float *sensorData = (float*)(payload);
          *payloadLen = USR_OUT_DATA1_PAYLOAD_LEN;
          GetAccelsData(accels);
          for (int i = 0; i < 3; i++, n++){
              sensorData[n] = (float)accels[i];
          }
          GetRatesData(rates);
          for (int i = 0; i < 3; i++, n++){
              sensorData[n] = (float)rates[i];
          }
          GetMagsData(mags);
        }
```

(continues on next page)

(continued from previous page)

```

        for (int i = 0; i < 3; i++, n++){
            sensorData[n] = (float)mags[i];
        }
    }
    break;
    // place additional user packet preparing calls here
    // case USR_OUT_XXXX:
    //     *payloadLen = YYYY; // total user payload length, including user_
    ↪packet type
    //     payload[0] = ZZZZ; // user packet type
    //     prepare data here
    //     break;
    case USR_OUT_TLT:
    {
        if ( tlt == NULL )
        {
            *payloadLen = 0;
            ret = FALSE;
        }
        else
        {
            // get results
            *payloadLen = sprintf((char*)payload,
                                "pitch:%.3f\troll:%.3f\tax:%.3f\tay:%.3f\taz:%.3f\n",
                                tlt[0], tlt[1], tlt[2], tlt[3], tlt[4]);
        }
    }
    break;

default:
    *payloadLen = 0;
    ret = FALSE;
    break;    /// unknown user packet, will send error in response
}

return ret;
}

```

This procedure will be called at the defined rate by the framework.

The framework default outputs calibrated IMU sensor data. To output your own packets, the user should tell the framework the packet code of your packet, and then feed the algorithm results to the encoding procedure we just implemented above.

- Register the user-defined packet in the framework.

This can be done by calling **setOutputPacketCode** when initializing user-defined algorithm in **dataProcessingAnd-Presentation.c**. To use **setOutputPacketCode**, the user need

```
#include "SystemConfiguration.h"
```

and then call it in

```

void initUserDataProcessingEngine()
{
    InitUserDataStructures();    // default implementation located in file UserData.c
    InitUserFilters();           // default implementation located in file UserFilters.
    ↪C

```

(continues on next page)

(continued from previous page)

```

    InitUserAlgorithm();           // default implementation located in file user_
    ↪algorithm.c
    setOutputPacketCode(0x7A32);   // set output packet to user defined packets
}

```

In this way, the default packet will be replaced by the user-defined packet.

- Feed algorithm results to the encoding procedure.

In **dataProcessingAndPresentation.c**, after calling the user-defined algorithm, the framework will call

```

WriteResultsIntoOutputStream(results) ;    // default implementation located in file_
    ↪file UserMessaging.c

```

to feed **results** to **UserMessaging.c**. **WriteResultsIntoOutputStream** is implemented like this:

```

void WriteResultsIntoOutputStream(void *results)
{
    // implement specific data processing/saving here
    tlt = (float*)results;
}

```

where **tlt** is a global variable declared as

```

static float *tlt; // pointer to algorithm results

```

Now, compile the project, upload the firmware, and the user can get results via the interface UART.

## 6.8 CAN J1939 Messaging

### CAN J1939 Example Application For OpenIMU330RI

- The example can be used as is or customized to suit the customer's system requirements.
- The SAE J1939 standards document set specifies the requirements for systems based on J1939 messaging. The SAE site provides a full list of the J1939 standard document set - [Link](#)
- In particular:
  - Section 3 of the SAE J1939 standards document provides the high-level technical requirements for systems that use J1939 messaging.
  - Section 5 of the SAE J1939-21 standards document provides the technical requirements for J1939 data link layer for all SAE J1939 applications.
  - The license for using an SAE standards document do not allow distribution of the documents. SAE J1939 documents can be purchased online at the IHS Standards Store - [Link](#)
  - There are many J1939 related documents available that can be freely distributed. We provide two such documents here:
    - \* Vector Informatik GmbH provides a document which is a good introduction to J1939 download link.
    - \* Kvaser provides a J1939 Overview document - download link.

---

**Note:** If you use any links here, user your browser back button to return

---

The following pages describe the CAN J1939 Example Application Details:

- VSCode project for the J1939 CAN Example Application
- Application Dataflow and Synchronization diagram
- Examples of the J1939 CAN messages implemented in the application.

### 6.8.1 VSCode project for the J1939 CAN IMU Example Application

The *IMU* project for OpenIMU300RI is the example which implements basic IMU functionality and transmits calibrated sensors data over CAN bus using J1939 protocol.

- The most important files are found in the bottom level 'include', 'include/API', 'lib/J1939/include', 'lib/J1939/src', 'src', and 'src/user' directories.
- These directories provide the user visible and modifiable files, including the example application code and the header files that provide the function prototypes for the user and library code and critical definitions.
- The directory structure and files are shown in the following screen capture from VSCode.

### 6.8.2 Example J1939 Application Diagrams

The following diagrams illustrate:

- The typical data processing flow in OpenIMU300RI applications

---

**Note:** An internal timer, set to provide a 200Hz tick, provides the basic timing synchronization for all task functions.

---

### 6.8.3 CAN J1939 Messages

In next chapter provided description of the J1939 messages used in OpenIMU300RI application examples. Users can keep the implemented messages as is, modify them, or add new messages.

The following message categories are used:

#### Requests

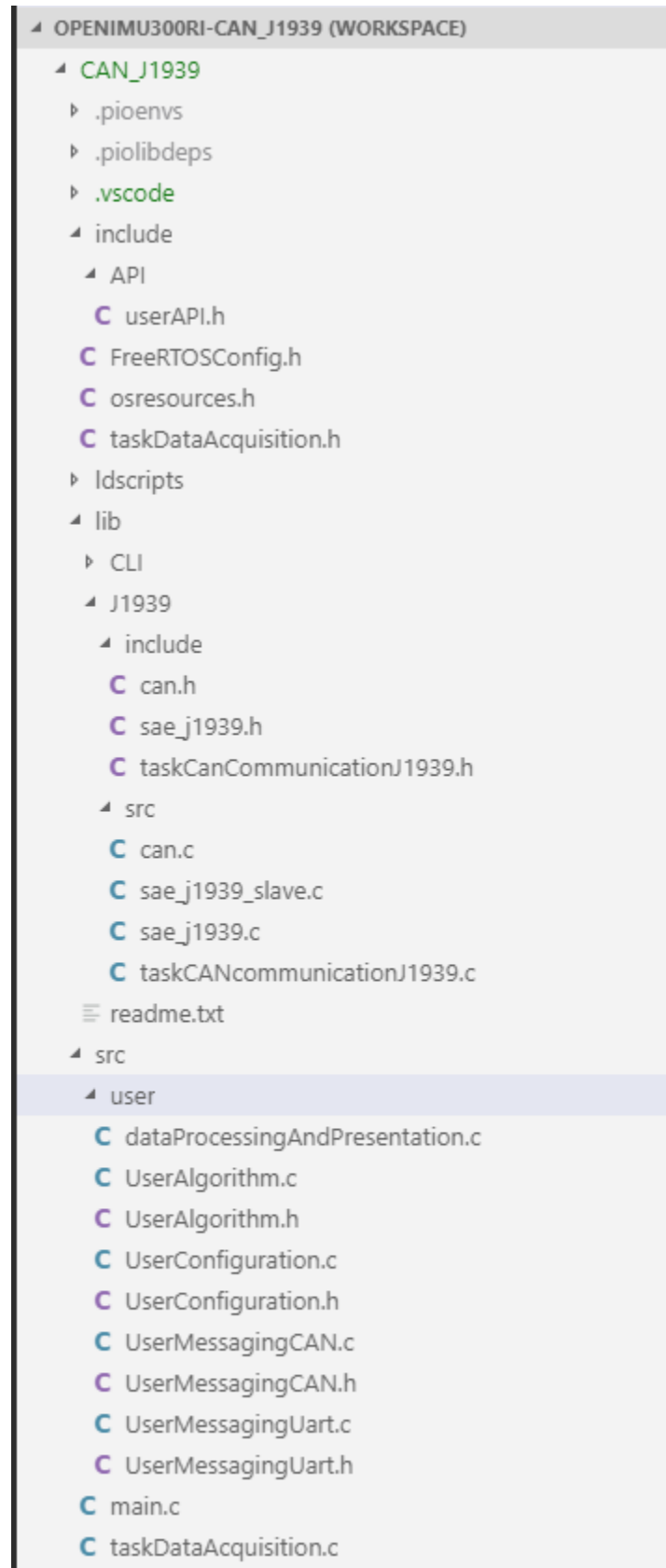
*Set Requests.* Set requests are used by Electronic Control Units (ECUs) to configure the OpenIMU300RI on the network.

*Get Requests.* Get requests are used for requesting information from the OpenIMU300RI. If the request is for the OpenIMU300RI, it will build and send a response packet to the requesting node.

#### Data Packets

Data packets are broadcast periodic messages with controllable rates, usually from 0 Hz (quiet mode) to 100Hz. The types of transmitted by OpenIMU300RI messages can be controlled by *Set Requests*. Also data packets can be arbitrary requested from OpenIMU by external ECUs using *Get Requests*.





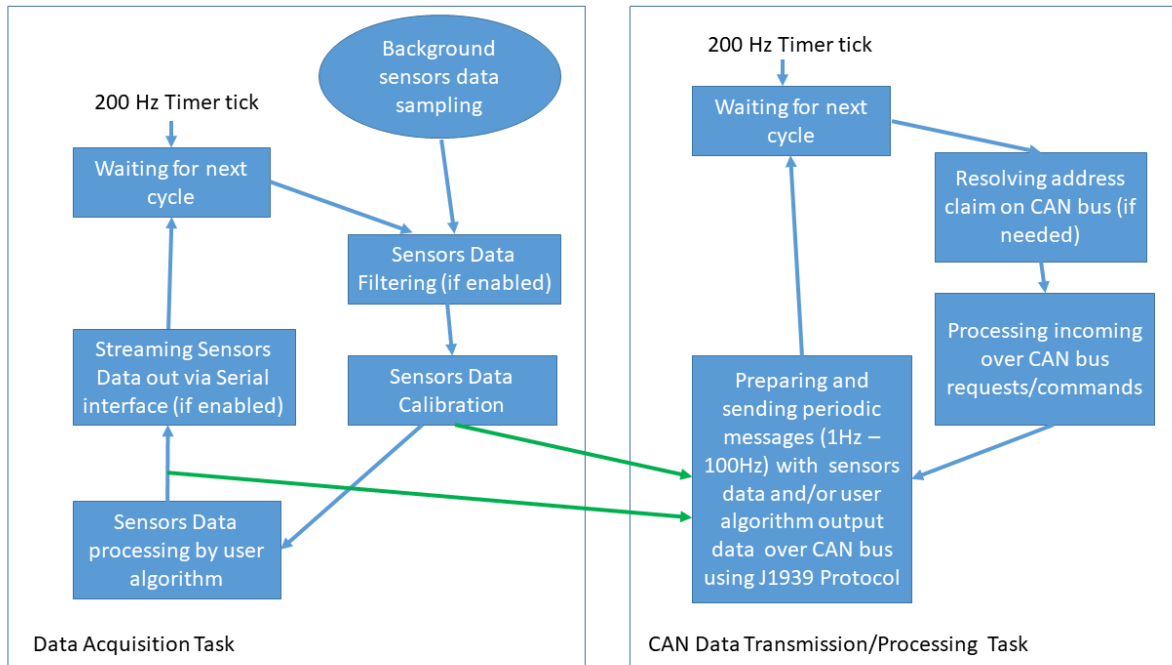


Fig. 2: J1939 Example Application data processing and events scheduling

## CAN J1939 Set Request Messages

### Set Commands

The following Set requests have been implemented in J1939 based application examples. Users can modify provided requests and/or implement their own unique commands.

Table 3: *Set Commands*

<b>Request</b>	<b>PF</b> (dec)	<b>PS</b> (dec)	<b>PGN</b>	<b>Payload Length</b> (bytes)	<b>Purpose</b>
<i>Save Configuration</i>	255	81	65361	3	Save all configuration data to non-volatile memory
<i>Reset Algorithm</i>	255	80	65360	3	Reset algorithm to initial conditions
<i>Mag Alignment</i>	255	94	65374	2	Mag alignment and status request
<i>Set Packet Rate Divider</i>	255	85	65365	2	Set rate dividers to increase/decrease rate packets are set
<i>Set Data Packet Type(s)</i>	255	86	65366	2	Select packet types to be sent periodically
<i>Set Digital Filters Cutoff Frequencies</i>	255	87	65367	3	Set LPF cutoff frequency for rate sensors and accelerometers
<i>Set Orientation</i>	255	88	65368	3	Set unit orientation
<i>Set Lever Arm (TBD)</i>	255	95	65375	8	Set unit Lever Arm (where applicable)
<i>Set Bank of PS Numbers for Bank0</i>	255	240	65520	8	Reconfigure PS numbers for set requests
<i>Set Bank of PS Numbers for Bank1</i>	255	241	65521	8	Reconfigure PS numbers for set requests

**Note:** PS values for all but the “Set Bank of PS Numbers for Bank0/Bank1” Set Commands can be changed by the the commands “Set Bank of PS Numbers” (see below). Updated values can be saved in nonvolatile memory and will be active upon following system restart/power-up. Provided in the table PS values are default values.

### Save Configuration

The next table provides the descriptions of the payload fields of the command and response messages.

Table 4: *Save Configuration Request/Response Payload Fields*

Byte	Description/Values
0	Type: 0 = Request, 1 = Response
1	Destination Address
2	Response: 0 = Fail, 1 = Succeed

### Reset Algorithm

The following table provides the descriptions of the payload fields of the command and response messages.

Table 5: *Reset Algorithm Request/Response Payload Fields*

Byte	Description/Values
0	Type: 0 = Request, 1 = Response
1	Destination Address
2	Response: 0 = Fail, 1 = Succeed

### Mag Alignment (INS Application Example)

The following tables provides the descriptions of the payload fields of the command and response messages.

Table 6: *Mag Alignment Request Payload Fields*

Byte	Description/Values
0	Destination Address
1	Commands: 0 - Status Request 1 - Start Alignment 5 - Confirm and Save

Table 7: *Payload Fields of 64 bit Response*

Bits	Description	Value
bits 0:7	Command	0 - Status request, 1 - Start alignment
bit 8:15	Alignment State	0 - Idle 12 - Alignment in process 11, 13 - Data Collection complete
bit 16:27	Estimated Hard Iron X Bias, Gauss	-8 G to +8 G , scale 1/256 G/bit, offset -8G
bits 28:39	Estimated Hard Iron Y Bias, Gauss	-8 G to +8 G , scale 1/256 G/bit, offset -8G
bits 40:49	Estimated Soft Iron Ratio	0 to 1 1/1024 per Lsb
bits 50:63	Estimated Soft Iron Angle	-3.14 to 3.14 RAD, scale 0.0015339, offset -3.14159

**Set Packet Rate Divider**

The following table provides the values of the packet rate divider response payload

Table 8: Set Packet Rate Divider Request/Response Payload Fields

Byte	Description	Byte Value
0	Destination Address	Unique
1	Packet Divider Value	<b>Byte Value - Packet Broadcast Rate (Hz)</b> 0 - Quiet Mode - no broadcast 1 - 100 (default) 2 - 50 4 - 25 5 - 20 10 (0x0a) - 10 20 (0x14) - 5 25 (0x19) - 4 50 (0x32) - 2

**Set Periodic Data Packet Type(s)**

The following table provides the *Set Data Packet Type(s)* payload. Each bit in the request payload enables specific data packet for periodic transmission. Any combination of data packets can be chosen.

Table 9: Set Data Packet Type(s) Field

Byte	Description	Byte Value
0	Destination Address	Unique
1	Selected Data Packet Type(s) Bitmask (LSB)	Data Packet Type(s) Bitmask: Bit 0 - SSI2 Bit 1 - Angular Rate Bit 2 - Acceleration Bit 3 - Magnetometer
2	Selected Data Packet Type(s) Bitmask (MSB)	Reserved

**Set Digital Filter Cutoff Frequencies**

The following table provides descriptions of the request payload

Table 10: *Digital Filter Cutoff Frequencies Request Payload*

Payload Byte	Description/Values	Values
0	Destination Address	Unique
1	Cutoff Frequencies (Hz) for Angular Rate Sensors	0, 2, 5, 10, 25, 40, or 50
2	Cutoff Frequencies (Hz) for Accelerometer Sensors	0, 2, 5, 10, 25, 40, or 50

### Set Orientation

The following table shows the payload layout

Table 11: *Set Orientation Payload Layout*

Byte	Meaning	Value
0	Destination Address	Unique
1	Orientation Value (MSB)	see table below
2	Orientation Value (LSB)	see table below

The following table provides the orientation values and meanings:

Table 12: *Set Orientation Field Descriptions*

Orientation Value	X/Y/Z Axis	Orientation Value(cont)	X/Y/Z Axis
0x0000	+Ux +Uy +Uz (default)	0x00C4	+Uz +Uy -Ux
0x0009	-Ux -Uy +Uz	0x00CD	-Uz -Uy -Ux
0x0023	-Uy +Ux +Uz	0x00D3	-Uy +Uz -Ux
0x002A	+Uy -Ux +Uz	0x00DA	+Uy -Uz -Ux
0x0041	-Ux +Uy -Uz	0x0111	-Ux +Uz +Uy
0x0048	+Ux -Uy -Uz	0x0118	+Ux -Uz +Uy
0x0062	+Uy +Ux -Uz	0x0124	+Uz +Ux +Uy
0x006B	-Uy -Ux -Uz	0x012D	-Uz -Ux +Uy
0x0085	-Uz +Uy +Ux	0x0150	+Ux +Uz -Uy
0x008C	+Uz -Uy +Ux	0x0159	-Ux -Uz -Uy
0x0092	+Uy +Uz +Ux	0x0165	-Uz +Ux -Uy
0x009B	-Uy -Uz +Ux	0x016C	+Uz -Ux -Uy

**Set Lever Arm (TBD)**

The following table shows the payload layout

Table 13: *Set Lever Arm payload*

Byte	Description
0	Destination Address
1	reserved
2	Wheel Distance Value (LSB), mm
3	Wheel Distance Value (MSB), mm
4	Lever Arm Bx Value (LSB), mm
5	Lever Arm Bx Value (MSB), mm
6	Lever Arm By Value (LSB), mm
7	Lever Arm By Value (MSB), mm

**Set Bank of PS Numbers**

The following tables provide descriptions of the payload for Bank0 and Bank1 set commands

Table 14: *Set Bank of PS Numbers for Bank0 Payload*

Byte	Parameters
0	Destination Address
1	Reset Algorithm PS number
2	Save Configuration PS number
3	Status Request PS number
4	Mag Align Command PS number
5-7	Reserved



Table 15: *Set Bank of PS Numbers for Bank1 Payload*

Byte	Parameters
0	Destination Address
1	Set Packet Rate PS number
2	Set Packet Type(s) PS number
3	Set Digital Filter Cutoff Frequencies PS number
4	Set Orientation PS Number
5	Set User Behavior PS Number
6	Set Lever Arm PS Number
7	Reserved

## CAN J1939 Get Request Messages

### Contents

- *Get Requests*
- *Responses to Get Requests*

## Get Requests

Get requests are used by other ECUs in the network to retrieve information from the OpenIMU300RI. All Get requests are formed as a Request message as specified earlier. The format and content of the Request message has next format:

Extended header:

PF : 234,  
 PS : 255,  
 DLC : 3,  
 Priority: 6,  
 PGN : 60159.

Table 16: *Request Payload*

Byte	Description
0	N/A
1	PF of requested parameter
2	PS of requested parameter

In table below provided list of the parameters which can be requested from ECU, including their PF, PS and payload length of response messages

Table 17: List of ECU parameters available for Requests

Parameter	PF (dec)	PS (dec) (See note)	Payload Length (bytes)
<i>Software Version</i>	254	218	5
<i>ECU ID</i>	253	197	8
<i>Packet Rate</i>	255	85	2
<i>Packet Type</i>	255	86	3
<i>Digital Cutoff Frequency</i>	255	87	3
<i>Orientation</i>	255	88	3
<i>Lever Arm(TBD)</i>	255	95	8

**Note:**

- Provided PS values for all but the *Get Software Version* and *Get ECU ID* can be changed by the “Set Bank of PS Numbers for Bank1” command. The given values are the default values.
- In responses values of PF and PS field in extended headers have the same PF+PS values as requested.

**Responses to Get Requests**

The following table describe the payloads for responses to Get Requests

Table 18: Software Version Response Payload

Byte	Description
0	Major Version Number
1	Minor Version Number
2	Patch Number
3	Stage Number
4	Build Number

Table 19: *ECU ID 64 Bit Response Payload*

Bits	Contents
bits 0	Arbitrary Address
bit 1:3	Industry Group
bit 4:7	Vehicle System Instance
bits 8:14	System Bits
bits 15	Reserved
bits 16:23	Function Bits
bits 24:28	Function Instance
bits 29:31	ECU Bits
bits 32:42	Manufacturer code
bits 43:63	ID bits

Table 20: *Packet Rate Response Payload*

Byte	Description
0	Source Address
1	Output Data Rate

Table 21: *Packet Type Response Payload*

Byte	Description
0	Source Address
1	Packet Types Bitmask (LSB)
2	Packet Types Bitmask (MSB)

Table 22: *Digital Cutoff Frequency Response Payload*

Byte	Description
0	Source Address
1	Acceleration Cutoff
2	Angular Rate Cutoff

Table 23: *Orientation Response Payload*

Byte	Description
0	Source Address
1	Orientation Value (MSB)
2	Orientation Value (LSB)

Table 24: *Lever Arm Response Payload (TBD)*

Byte	Description
0	Source Address
1	reserved
2	Wheel Distance Value (LSB), mm
3	Wheel Distance Value (MSB), mm
4	Lever Arm Bx Value (LSB), mm
5	Lever Arm Bx Value (MSB), mm
6	Lever Arm By Value (LSB), mm
7	Lever Arm By Value (MSB), mm

---

**Note:**

- For Orientation, Cutoff Frequencies Packet Type and Packet Rate responses values of parameters will be the same as in the set commands for these parameters.
- 

### CAN J1939 Data Messages

The following Data messages are implemented in the example applications. The user can modify provided messages or add messages as needed. The rate of data messages can be configured by SET commands.

Table 25: *Data Messages*

<b>Data Packet</b>	<b>PF (dec)</b>	<b>PS (dec)</b>	<b>PGN (dec)</b>	<b>Data Length (bytes)</b>	<b>Purpose</b>
Slope Sensor Information Type 2	240	41	61481	8	Provide high accuracy pitch & roll rates
Angular Rate Sensor Data	240	42	61482	8	Provide moderate accuracy pitch, roll and yaw rates
Acceleration Sensor Data	240	45	61485	8	Provide moderate accuracy X, Y, and Z axes acceleration
Magnetometer Sensor Data	255	106	65386	8	Provide readings from magnetic sensor for X, Y, and Z axes

**Slope Sensor Information - Type 2 (SSI2) Data Packet**

The following table describes the SSI2 Data Packet Payload:

Table 26: *SSI2 Data Packet Payload*

<b>Bytes</b>	<b>Field Name</b>	<b>Range</b>	<b>Resolution</b>	<b>Offset</b>
0:2	Pitch	-250 to +252 deg	1/32768 deg/bit	-250 deg
3:5	Roll	-250 to +252 deg	1/32768 deg/bit	-250 deg
6:7	FoM, Latency	Ignore	Ignore	Ignore

---

**Note:** SSI2 Data Packet is applicable for VG-AHRS or INS Applications only.

---

### Angular Rate Data Packet

The following table describes the Angular Rate Data Packet:

Table 27: *Angular Rate Data Packet Payload*

Byte Number	Parameter	Range	Resolution	Offset
0:1	Angular Rate X	-250 to +252 deg/s	1/128 deg/second/bit	-250 deg
2:3	Angular Rate Y	-250 to +252 deg/s	1/128 deg/second/bit	-250 deg
4:5	Angular Rate Z	-250 to +252 deg/s	1/128 deg/second/bit	-250 deg
FoM,Latency	FoM,Latency	Ignore	Ignore	Ignore

### Acceleration Data Packet

The following table describes the Acceleration Data Packet:

Table 28: *Acceleration Data Packet Payload*

Byte Number	Parameter	Range	Resolution	Offset
0:1	Acceleration X	-320 to 320/55 m/s**2	0.01 m/s**2/bit	-320 m/s**2
2:3	Acceleration Y	-320 to 320/55 m/s**2	0.01 m/s**2/bit	-320 m/s**2
4:5	Acceleration Z	-320 to 320/55 m/s**2	0.01 m/s**2/bit	-320 m/s**2
6:7	FoM,Latency	Ignore	Ignore	Ignore

### Magnetometer Data Packet

The following table describes the Magnetometer Data Packet:

Table 29: *Magnetometer Data Packet Payload*

Byte Number	Parameter	Range	Resolution	Offset
0:1	Magnetic Field X	-8 to 8 Gauss	4000 LSB/G	-8 Gauss
2:3	Magnetic Field Y	-8 to 8 Gauss	4000 LSB/G	-8 Gauss
4:5	Magnetic Field Z	-8 to 8 Gauss	4000 LSB/G	-8 Gauss
6:7	FoM, Latency	Ignore	Ignore	Ignore

**Note:** As with all multiple byte fields, the LSB of each of the Data Packet fields is transmitted first.

---

### Algorithm Simulation System

---

**GNSS-IMU-SIM** is an IMU simulation project, which generates reference trajectories, IMU sensor output, GPS output, odometer output and magnetometer output. Users choose/set up the sensor model, define the waypoints and provide algorithms, and **gnss-imu-sim** can generate required data for the algorithms, run the algorithms, plot simulation results, save simulation results, and generate a brief summary.

GitHub Link: [GNSS-INS-SIM](#)

Use the browser's back button to return.



---

## Python Serial Driver

---

### Contents

- *OpenIMU Python Drivers*
- *Python Install*

## 8.1 OpenIMU Python Drivers

The OpenIMU Python driver supports communication with the hardware for data logging and device configuration over the main user UART interface of the OpenIMU hardware. When run in server mode, it allows connection of the OpenIMU with the developer's website Aceinna Navigation Studio and its friendly GUI interface.

You can start the OpenIMU server from GitHub Source code

GitHub Link: [python-openimu](#)

## 8.2 Python Install

Please install either Python 2.7 or 3.8 onto your PC & follow Readme file Instructions to Install all dependencies.

- Python Link: [Python-Download](#)
- Readme Link: [Install-Dependencies](#)

---

**Note:** Use the browser's back button to return to the OpenIMU documentation

---

## **Part II**

# **Products**

---

### OpenIMU300ZI - *EZ Embed* Industrial Module

---

#### Contents

- *Specifications*
- *Interfaces*
- *Pinout*
- *Eval Kit*
- *Ready to Use Application*

The following image shows the OpenIMU300ZI unit.



The OpenIMU300ZI *EZ Embed* module integrates highly-reliable MEMS inertial sensors (acceleration, angular rate/gyro, and magnetic field) in a miniature factory-calibrated package to provide consistent performance through the extreme operating environments.

OpenIMU300ZI has excellent acceleration and gyro performance that matches systems ten times more expensive. It is easy to synchronize and interface with external GPS, as well as other sensors.

- Integrated 3-Axis Angular Rate
- Integrated 3-Axis Accelerometer
- Integrated 3-Axis Magnetic Sensor
- 168MHz STM32 M4 CPU
- SPI / UART Interfaces
- Max ODR 200Hz
- Synchronization Input
- In-System Upgrade
- Small Size (24x37x9.5mm)
- Drop-in Upgrade for IMU380ZA, IMU381ZA
- Wide Temp Range -40 to 85 ° C
- High Reliability > 50,000hr MTBF

## 9.1 Specifications

### 9.1.1 Environmental, Electrical, and Physical Specifications

#### ENVIRONMENT

Specifications	
Operating Temperature (°C)	-40 to 105
Non-Operating Temperature (°C)	-65 to 150
Enclosure	

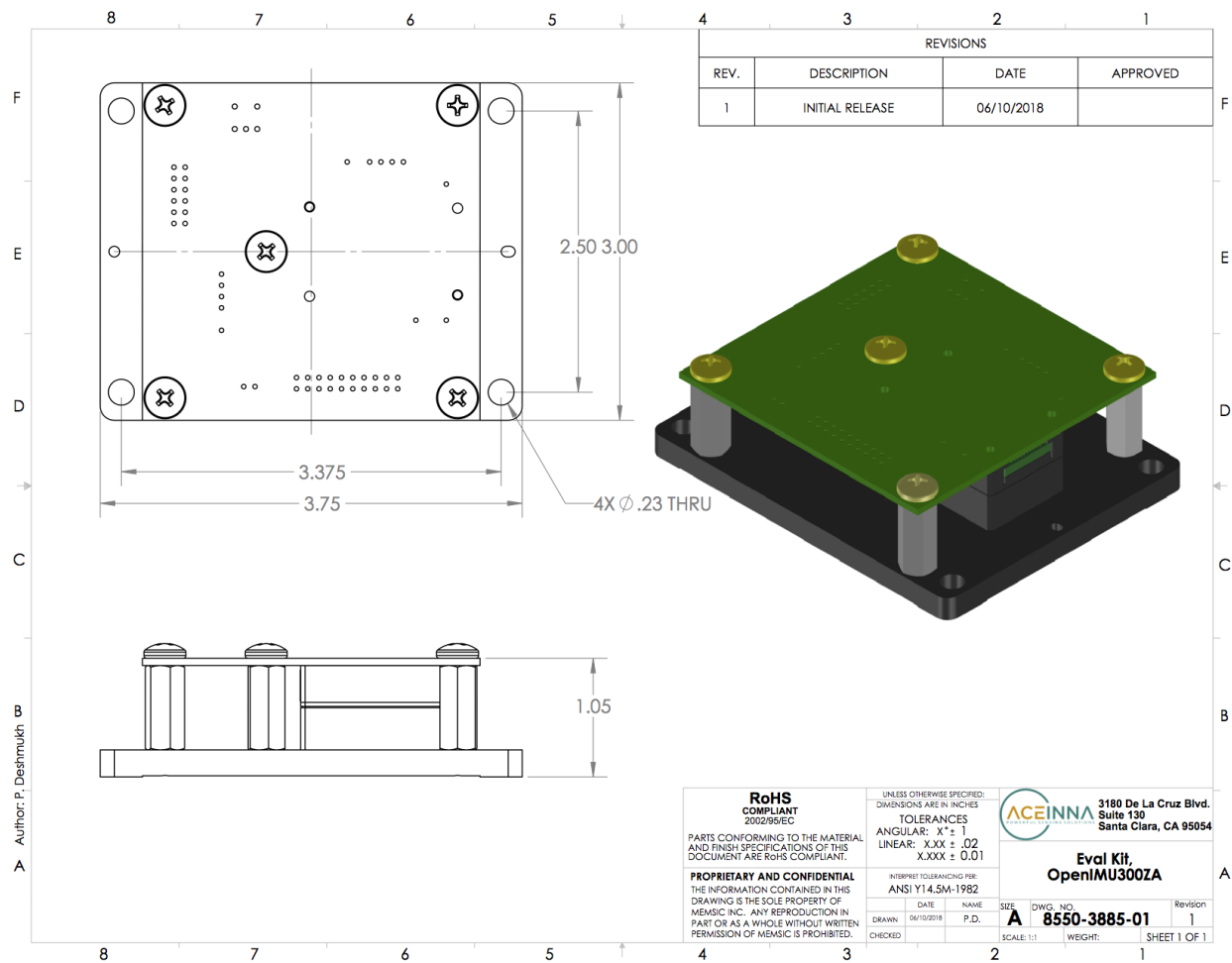
#### ELECTRICAL

Specifications	
Input Voltage (VDC)	3.0 - 5.5
Power Consumption (mW)	< 250
Digital Interface	SPI or UART
Output Data Rate	up to 200Hz (SPI) up to 100Hz (UART)
Input Clock Sync	1KHz pulse (Configurable)

#### PHYSICAL

Specifications	
Size (mm)	24.15 x 37.70 x 9.50
Weight (gm)	< 17
Interface Connector	20-Pin (10x2) 1.0mm pitch header

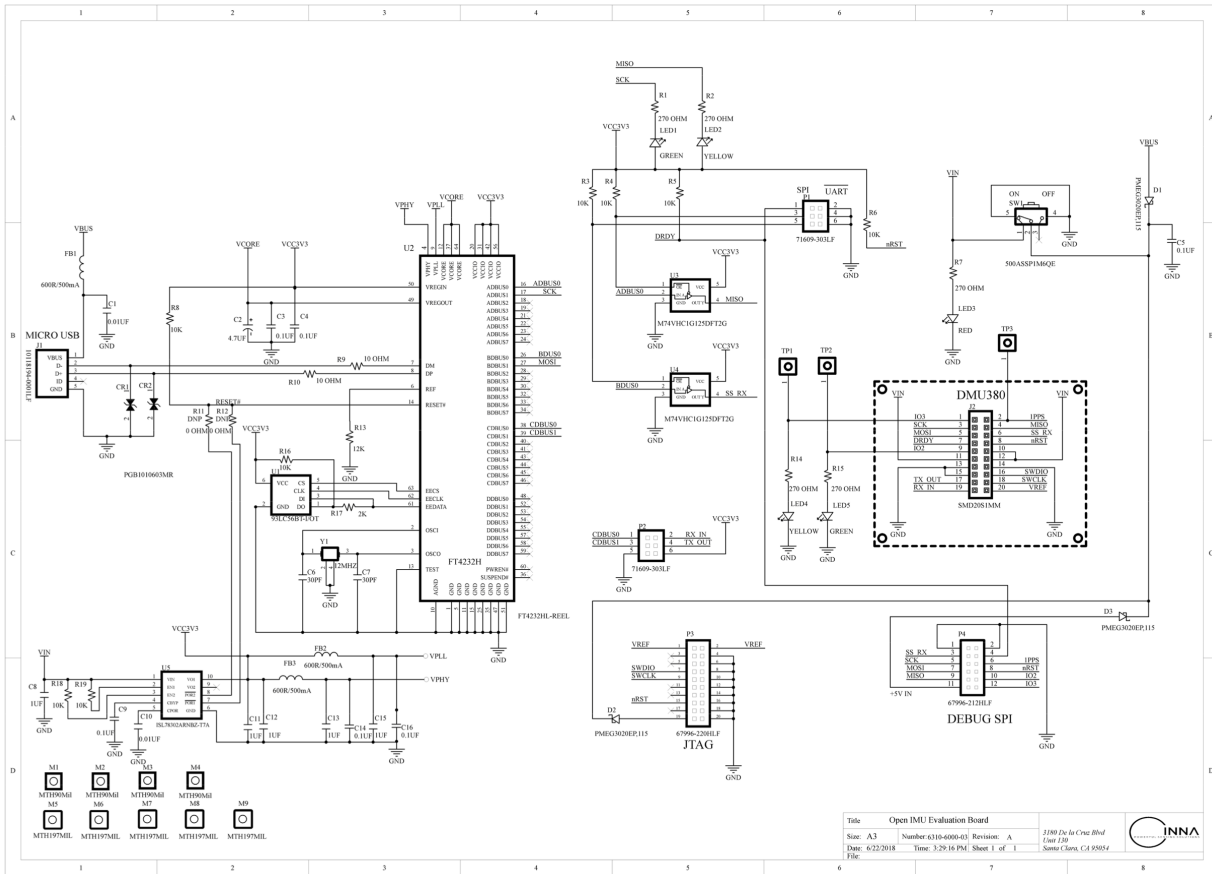
9.1.2 OpenIMU300ZI EVK Mechanical Drawing



**Note:** Use the browser's back button to return to this page.

[Mechanical Drawing download link](#)

### 9.1.3 EVB Schematic



Schematic download

## 9.2 Interfaces

### 9.2.1 SPI and UART

#### Contents

- *Ports*
- *SPI & UART Messaging*

#### Ports

The OpenIMU300ZI can be configured in a number of ways for communication with external world. There are up to three external UART ports and one external SPI port.

Typical configurations include:

<b>3 UART Mode</b>	<ul style="list-style-type: none"> <li>• User UART</li> <li>• GPS/External Sensor UART</li> <li>• Debug UART</li> </ul>
<b>UART + SPI Mode</b>	<ul style="list-style-type: none"> <li>• User SPI Port</li> <li>• GPS/Debug UART</li> </ul>

## SPI & UART Messaging

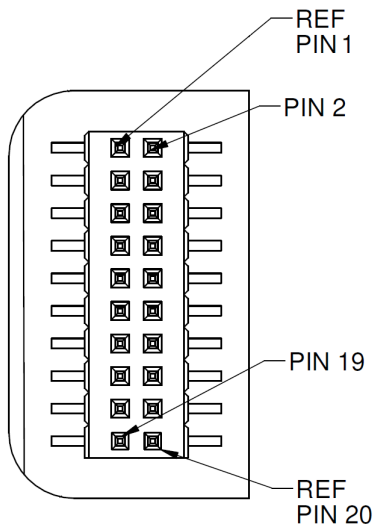
To learn more about the OpenIMU SPI & UART Messaging Framework, please see the following pages:

1. [SPI Messaging Framework](#)
2. [UART Messaging Framework](#)

## 9.3 Pinout

### 9.3.1 Connector Pinout - Including GPS Sensor Interface

The OpenIMU300ZI main connector is a SAMTEC FTM-110-02-F-DV-P defined below. The mating connector that pairs with the main connector is the SAMTEC CLM-110-02.



OpenIMU300ZI Interface Connector

J2 is 20-pin connector and it used for connecting the OpenIMU300ZI unit into Open IMU evaluation board. The connector pin definitions are defined in the table below. The GPS-related signals are noted.

#### Interface Connector Pin Definitions



Pin	Main Function	Alternative Function
1	GPIO ( IO3 )	Output by default
2	Synchronization Input	GPS 1PPS Input
3	User UART TX (Output) (Serial Channel 0 )	SPI Clock (SCLK) Input
4	User UART RX (Input) (Serial Channel 0)	SPI Data (MISO) Output
5	UART1 TX (Output) (Serial Channel 1)	SPI Data (MOSI) Input
6	UART1 RX (Input) (Serial Channel 1)	SPI Chip Select (SS) Input
7	SPI/UART Interface Selector	Data Ready (SPI) Active edge falling
8	External Reset (NRST))	
9	GPIO ( IO2 )	Output by default
10	Power VIN (3-5 VDC)	Power VIN (3-5 VDC)
11	Power VIN (3-5 VDC)	Power VIN (3-5 VDC)
12	Power VIN (3-5 VDC)	Power VIN (3-5 VDC)
13	Power GND	Power GND
14	Power GND	Power GND
15	Power GND	Power GND
16	SWDIO (SWD debug interface)	
17	UART2 TX (Serial Channel 2)	Debug interface GPS
18	SWCLK (SWD debug interface)	
19	UART2 RX (Serial Channel 2)	Debug Interface GPS
20	Reference voltage for SWD debug interface	

### Power Input and Power Input Ground

Power is applied to the OpenIMU300ZI on pins 10 through 15. Pins 13-15 are ground; Pins 10-12 accepts 3 to 5 VDC unregulated input. Note that these are redundant power ground input pairs.

---

**Note:** Do not reverse the power leads or damage may occur. Do not add greater than 5.5 volts on the power pins or damage may occur. This system has no reverse voltage or over-voltage protection.

---

---

**Note:** Serial channel functions can be arbitrary assigned in the FW. Default assignments are:

---

Serial channel 0 -> USER UART (dedicated for user messages).

Serial channel 1 -> GPS UART (dedicated for connecting external GPS).

Serial channel 2 -> DEBUG UART (dedicated for debug messages and CLI interface).

In some application examples (INS, VG\_AHRS) in file main.c performed reassignment of serial channels to different functions.

---

---

**Note:** Pin 7 needs to be grounded (LOW) upon unit startup to force unit into UART interface mode. To force unit into SPI mode this pin needs to be either unconnected or connected to the input or external device (can be externally pulled UP via 10K resistor).

**In SPI mode only serial channel 2 available and can be used for communication with GPS or as DEBUG channel.**

---

## 9.3.2 ARM Cortex CPU

The OpenIMU300ZI uses ST's powerful Cortex M4 series of Microcontrollers.

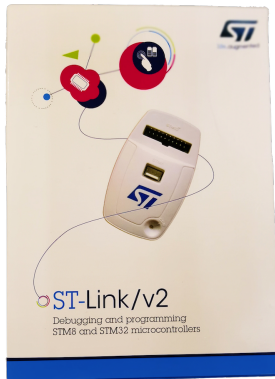
- FPU
- DSP instructions
- 1MByte Flash
- 192KB SRAM
- 168 MHz
- Rich Set of peripherals

Learn more about OpenIMU300ZI's CPU at <http://www.st.com/en/microcontrollers/stm32f405rg.html>.

## 9.4 Eval Kit

### 9.4.1 OpenIMU300ZI Eval Kit

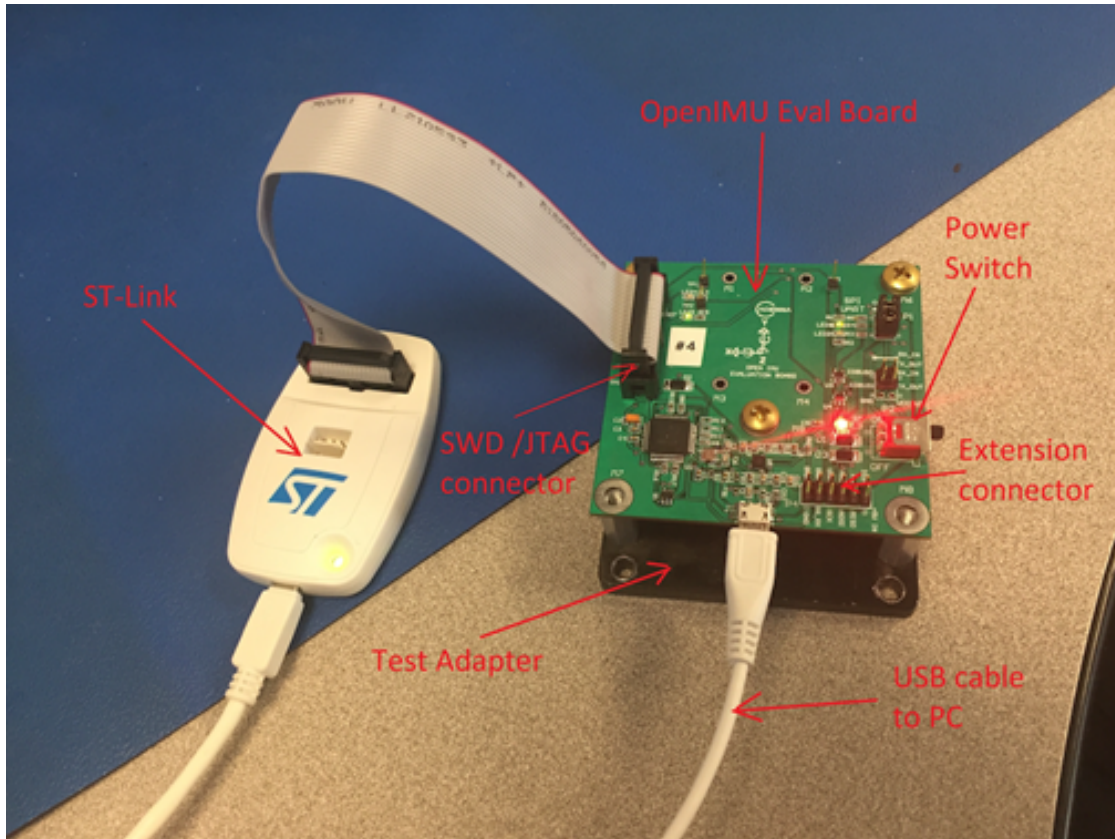
The OpenIMU300ZI evaluation kit consists of a robust and easy-to-use eval board, a test fixture, the OpenIMU300ZI IMU module, and an ST-LINK J-TAG pod.



## Overview

### 1. Introduction

The OpenIMU evaluation kit is a hardware platform to evaluate the OpenIMU300ZI inertial navigation system and develop various applications based on this platform. Supported by the Aceinna Navigation Studio the kit provides easy access to the features OpenIMU300ZI and explains how to integrate the device in a custom design. The OpenIMU evaluation kit include OpenIMU300ZI, evaluation board with various interface connectors and test adapter for mounting OpenIMU300ZI unit.



## 2. Components

- OpenIMU Evaluation board, which includes:
  - Virtual COM-port USB interface, providing connectivity to OpenIMU300ZI unit from PC
  - Connector for programming and debugging target via Serial Wire Debug (SWD) interface
  - Connector for interfacing OpenIMU300ZI from custom-designed system.
  - Test terminals for connecting oscilloscope or logic analyzers to the dedicated OpenIMU300ZI signals.
- OpenIMU300ZI unit. Please note, that it installed on the bottom side of evaluation board.
- Test fixture adapter for convenient aligned mounting of OpenIMU evaluation board and OpenIMU300ZI unit
- ST-Link debugger for in-system development of application code

### 2.1 OpenIMU300ZI unit

OpenIMU300ZI is 9 DOF (degrees of freedom) fully calibrated inertial unit. It is used as the base for development custom inertial navigation applications.

### 2.2 OpenIMU Evaluation board

OpenIMU Evaluation board designed to provide convenient way for communicating with OpenIMU300ZI unit from PC, to expose serial and SPI interfaces to developer and to debug applications using ST-Link debugger vis SWD interface.

### 2.3 OpenIMU test adapter

OpenIMU test adapter used to firmly secure OpenIMU300ZI unit and Open IMU evaluation board in precisely aligned position.

## 2.4 ST-Link debugger

St-Link debugger is standard debugger provided by STMicroelectronics company. It used for in-system debugging of applications via SWD interface.

## 3. Open IMU evaluation board Headers and Connectors

### 3.1 Connector for plugging in OpenIMU300ZI unit (J2).

J2 is 20-pin connector and it used for connecting the OpenIMU300ZI unit into Open IMU evaluation board. The pin functions are described in the table on the “OpenIMU Modules » OpenIMU300ZI - EZ Embed Automotive Module » Connector Pinout - Including GPS Sensor Interface” page accessible from the Contents bar on the left.

### 3.2 Extension Header (P4)

OpenIMU evaluation board has 12-pin extension header. It designed to expose IMU interface signals to external system. The extension header pin functions described in table below

Pin	Main Function	Alternative Function
1	Power GND	Power GND
2	Power GND	Power GND
3	Serial Channel 1 RX (Input)	SPI Chip Select (SS) (Input)
4	IMU Data Ready (SPI interface Mode)	GPIO (UART interface mode)
5	User UART TX (Serial Channel 0) (Output)	SPI Clock (SCK) (Output)
6	Synchronization Input	1PPS Input from GPS
7	Serial Channel 1 TX (Output)	SPI Data (MOSI) (Input)
8	External Reset (NRST))	
9	User UART RX (Serial Channel 0) (Input)	SPI Data (MISO) (Output)
10	GPIO Output (IO2)	GPIO Input
11	Power VIN 5 VDC	Power VIN 5 VDC
12	GPIO Output (IO3)	GPIO Input

### 3.4 IMU interface type selection header (P1).

**Pins 1-2** define IMU **Interface Mode**:

If there is no connection between pins 1 and 2 (jumper is OFF) - **SPI** mode.  
if there is connection between pins 1 and 2 (jumper is ON) - **UART** mode (default).

**In SPI mode:**

**Jumpers between pins 3-4 and 5-6 need to be taken OFF** to prevent interference between SPI bus signals (SS and MISO) and serial interface signals from FTDI chip.  
IMU SPI interface signals (MISO, MOSI, SS, SCK, DRDY) routed to header P4.

---

**Note:** On **SPI** interface IMU acts as a **SLAVE** device.

---

---

**Note:** Not all provided application examples support SPI interface mode. Please refer to specific example for details.

---

**In UART mode:**

Jumper between pins **3-4** should be **“ON”** (default) if IMU **Serial Channel 0** ( USER main channel ) needs to be routed to PC via USB connection (on first in the row enumerated USB virtual COM port. See p.6).

Jumper between pins **3-4** should be **OFF** if IMU **Serial Channel 0** needs to be accessed from P2 connector.

Jumper between pins **5-6** should be **ON** (default) if IMU **Serial Channel 1** needs to be routed to PC via USB connection (on second in the row enumerated USB virtual COM port. See p.6).

Jumper between pins **5-6** should be **OFF** if IMU **Serial channel 1** needs to be accessed from P2 connector. For example if Serial Channel 1 used for connection with some external device (GPS or other)

### 3.5 IMU Serial Channel 2 mode selection header (P2).

Jumpers between pins **1-2** and **3-4** should be **ON** if IMU **Serial Channel 2** needs to be routed to PC via USB connection, for example in case of using IMU Serial Channel 2 for streaming out debug information to PC or as CLI interface (on third in the row enumerated USB virtual COM port. See p.6).

Jumpers between pins **1-2** and **3-4** should be **OFF** if IMU **Serial Channel 2** needs to be routed to some external device (for example GPS). In this case **pin 2 is RX** (to IMU) and **pin 4 is TX** (from IMU).

### 3.6 SWD (JTAG) connector (P3).

20-pin connector P3 used for connecting ST-Link or J-Link debuggers to the IMU for in-system debugging of applications via SWD interface. It has standard pin-out.

Pin	Main Function
1, 2	Vref
4, 6, 8, 10, 12, 14, 16, 18, 20	GND
7	SWDIO
9	SWCLK
15	nRST
19	3.3V from debugger

### 3.7 USB connector (J3)

USB connector used for powering up the IMU and evaluation board. Also its used to providing connectivity from PC to IMU via virtual serial ports. Up to 3 exposed IMU serial interfaces can be routed to PC.

## 4. OpenIMU evaluation board LED indicators

Evaluation board has few LED indicators for visual monitoring of data traffic on serial ports:

**LED2** indicator reflects activity on RX line of IMU main (user) serial interface (traffic to IMU)

**LED1** indicator reflects activity on TX line of IMU main (user) serial interface (traffic from IMU)

**LED3** indicator while lit indicates presence of the power (in case switch SW1 is “ON”)

**LED4** indicator reflects activity on GPIO3 (lit if high)

**LED5** indicator reflects activity on GPIO2 (lit if high)

## 5. Open IMU evaluation board power

Power to OpenIMU evaluation board provided by USB. To power system up - connect USB cable to connector J1 and turn “ON” switch SW1.

## 6. Communication with IMU from PC

The OpenIMU evaluation board has an FTDI chip FT4232 installed. This chip provides 4 virtual serial ports. When evaluation board set up to force IMU interface in UART mode (see p.3.4) up to 3 serial ports on IMU can communicate with PC. When evaluation board connected to PC and power switch turned “ON” in Device Manager board will appear as **4 new consecutive virtual COM ports**.

First in a row virtual port is routed to IMU’s main UART channel (Serial channel 0) (pins 3 and 4 on J2), and usually dedicated for sending commands to IMU and capturing responses and periodic messages from IMU. It usually used by python driver to establish communication between IMU and Aceinna Navigation Studio.

Second in a row virtual port routed to IMU’s Serial Channel 1 (pins 5 and 6 on J2) and potentially can be used for modeling or cloud data processing - sending GPS messages from PC to IMU and back.

Third in a row virtual port routed to IMU's Serial channel 2 (pins 17 and 19 on J2) and usually used as a debug/CLI serial channel .

## 9.4.2 OpenIMU300ZI Evaluation Kit Setup

To set up OpenIMU300ZI evaluation kit for development you'll need to perform next steps:

1. Unpack OpenIMU300ZI evaluation kit.
2. Push power switch to "OFF" position.
3. Connect OpenIMU300ZI evaluation board to the PC via USB cable. USB connection provides power to the test setup as well as connectivity between PC and IMU serial ports.
4. Connect ST-Link debugger to the PC via USB cable.
5. Connect OpenIMU300ZI evaluation board to ST-Link debugger using provided 20-pin flat cable.
6. Push power switch to "ON" position.

Now you are ready to debug and test your application.

- The following activities are addressed in the "Tools/Development Tools" section:
  - Download App with JTAG
  - Debugging with PlatformIO Debugger and JTAG Debug Adapter
  - Graphing & Logging IMU Data using the Acienna Navigation Studio

### OpenIMU Evaluation Kit Important Notice

This evaluation kit **is** intended **for** use **for** FURTHER ENGINEERING, DEVELOPMENT, DEMONSTRATION, OR EVALUATION PURPOSES ONLY. It **is not** a finished product **and** may **not**   
 → (yet)   
 comply **with** some **or any** technical **or** legal requirements that are applicable to   
 → finished products,   
 including, without limitation, directives regarding electromagnetic compatibility,   
 → recycling (WEEE),   
 FCC, CE **or** UL (**except as** may be otherwise noted on the board/kit). Aceinna supplied   
 → this board/kit   
 "AS IS," without **any** warranties, **with** all faults, at the buyer's and further users'   
 → sole risk. The   
 user assumes **all** responsibility **and** liability **for** proper **and** safe handling of the   
 → goods. Further,   
 the user indemnifies Aceinna **from all** claims arising **from the** handling **or** use of the   
 → goods. Due to   
 the **open** construction of the product, it **is** the user's **responsibility to take any and**   
 → **all appropriate**   
 precautions **with** regard to electrostatic discharge **and any** other technical **or** legal   
 → concerns.   
 EXCEPT TO THE EXTENT OF THE INDEMNITY SET FORTH ABOVE, NEITHER USER NOR ACEINNA   
 SHALL BE LIABLE TO EACH OTHER FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR   
 CONSEQUENTIAL DAMAGES.   
 No license **is** granted under **any** patent right **or** other intellectual **property** right of   
 → Aceinna covering   
**or** relating to **any** machine, process, **or** combination **in** which such Aceinna products **or**   
 → services might   
 be **or** are used.



### 9.4.3 OpenIMU300ZI Eval Board & Coordinate Frame

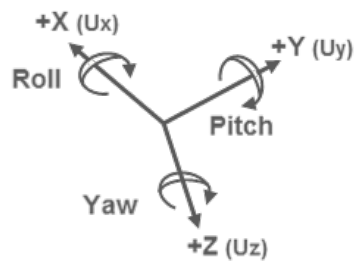
#### OpenIMU300ZI Eval Board and Fixture



The OpenIMU300ZI Eval Board is attached to a fixture for easy handling and isolation of the back side of the board from any contact. The EVB interfaces to the main connector of the OpenIMU300EZ as well as the OpenIMU330 evaluation module. The EVB and IMU module are mounted together to a precision fixture to assist in testing. The OpenIMU300EVB uses an FTDI 4-port Serial-to-USB converter to allow you to communicate with between the OpenIMU serial ports and a laptop computer. There are also jumper connections to use to connect to the device's primary SPI port. Use the JTAG interface to directly download compiled code to the device quickly.

#### OpenIMU300ZI Eval Module Default Coordinate System

The OpenIMU default coordinate systems is shown below. In the reference IMU apps, a configuration setting is provided to control the coordinate system. These configurable elements are known as **Configuration Parameters**.



## 9.5 Ready to Use Application

To learn about Ready-to-Use-Apps information & available for immediate download to your OpenIMU, please see the following pages:

1. [Ready-to-Use-Applications Information](#)
2. [Need to run the OpenIMU server before running one of the ready to use applications](#)
3. [Then upload a prebuilt app to your OpenIMU](#)

---

### OpenIMU300RI - Rugged Industrial CAN Module

---

#### Contents

- *Specifications*
- *Interfaces*
- *Pinout*
- *Eval Kit*
- *Ready to Use Application*

The following image shows the OpenIMU300RI unit.



The following diagram shows the default coordinate frame for the OpenIMU300RI. The coordinate frame can be changed using a UART or CAN message.

The OpenIMU300RI Robust Industrial CAN module integrates highly-reliable MEMS inertial sensors (acceleration, angular rate/gyro, and magnetic field) in a miniature factory-calibrated package to provide consistent performance through the extreme operating environments.

OpenIMU300RI has excellent acceleration and gyro performance that matches systems ten times more expensive.

- Hardware
  - Precision 3-axis MEMS Accelerometer
  - Low-Drift 3-axis MEMS angular rate sensor
  - High Performance 3-axis AMR Magnetometer
  - 168 MHz ARM M4 microcontroller
  - Wide Temp Range, -40C to +85C
  - Wide Supply Voltage Range, 5 V – 32 V
  - High Reliability, MTBF > 50k hours
  - IP67 Ampseal Connector
    - \* CAN 2.0 interface
  - UART - conditionally, one of the following:
    - \* Debug Console interface
    - \* -or- Aceinna Navigation Studio interface
  - SPI and I2C buses for communicating with internal sensor peripherals

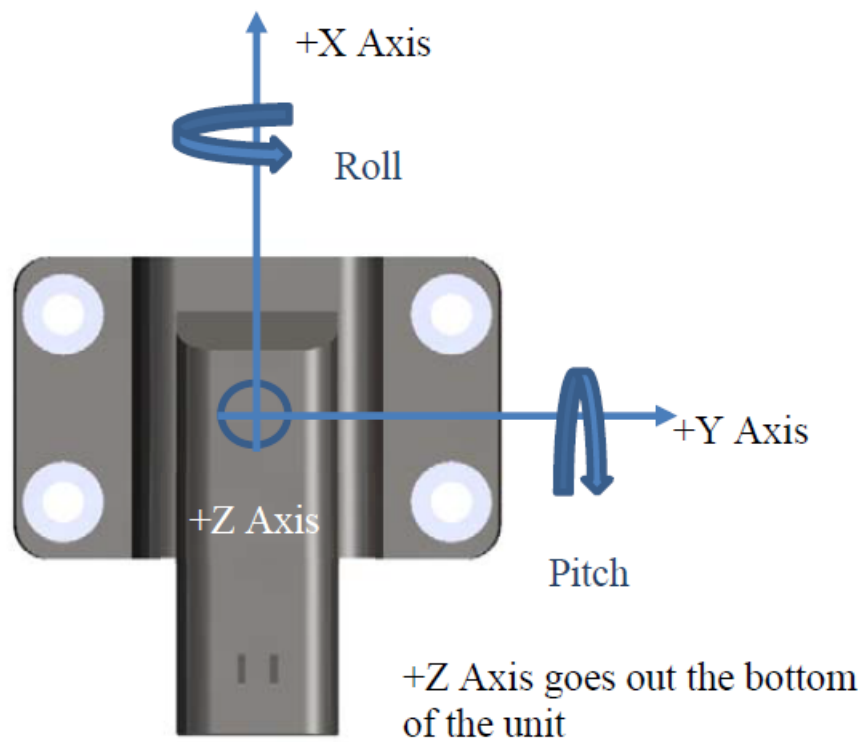


Fig. 1: OpenIMU300RI Default Coordinate Frame

- \* SPI - Angular Rate sensor
- \* I2C - Accelerometer and Magnetometer (if present)
- Firmware and Firmware Support
  - In-System Firmware Upgrade
  - Open Source Tool Chain
  - Open Source Algorithms (VG / AHRS / INS)
  - Built in 16-State Open Source Extended State Kalman Filter
  - Open Community & Support

## 10.1 Specifications

### 10.1.1 Electrical, and Physical Specifications

#### Performance Specification

Ta = 25°C, VDC = 12V, unless otherwise stated

Angular Rate	MIN	TYP <sup>2</sup>	MAX
Range (°/s)	-400		+400
Bias Instability (°/hr) <sup>1</sup>		6	
Bias Stability over Temp (°/s)		0.3	
Scale Factor Accuracy (%)		0.03	
Cross-Axis Error (%FSR)		0.02	
Angle Random Walk (°/√hr) <sup>1</sup>		0.3	
Configurable Bandwidth (Hz)	5		50
Acceleration	MIN	TYP <sup>2</sup>	MAX
Range (g)	-8		+8
Bias Instability (μg) <sup>1</sup>		10	
Bias Stability over Temp (mg)		3	
Scale Factor Accuracy (%FSR)		0.03	
Non-Linearity (%FSR)		0.03	
VRW (m/s/√hr) <sup>1</sup>		0.06	
Configurable Bandwidth (Hz)	5		50
Magnetic Field	MIN	TYP <sup>2</sup>	MAX
Range (mGauss)	-8000		+8000
Resolution (mGauss)		0.3	
Noise (mGauss/√Hz)		0.25	
Bandwidth (Hz)		5	

Note 1: Allan variance curve, constant temperature

Note 2: Typical values are 1-sigma values unless otherwise noted

#### Electrical Specifications

Characteristic	Specification
Input voltage	4.9 – 32 V
Over voltage	36 V
Reverse voltage	-36 V
Current	< 100 mA
Power	< 400 mW
Reset response	Automatic after voltage dropout
Start-up time	<2 seconds
Max Output Data Rate	100 Hz
CAN Baud rate	250k – 1M
RS232 Baud Rate	38400 – 230400

#### Physical Specifications

Characteristic	Specification
Dimensions	65 x 66 x 27 mm
Weight	< 75 g
Interface Connector	Ampseal 16 – 6 Position IP67
Mating Connector	TE Connectivity 776531-1

#### Environmental Specifications

Characteristic	Specification
Operating Temperature	-40 – 85 °C
Storage Temperature	-40 – 85 °C
Ingress Protection	IP67, IP69K

### 10.1.2 OpenIMU300RI Mechanical Diagram and Mounting Specifications

The following diagram shows the mechanical drawings for the OpenIMU300RI. The mechanical dimensions are in mm.

**Note:** Mounting Specifications

- Use 4 - M5 Alloy Steel Socket Head Screws to secure the OpenIMU300RI
- Torque the screws to 2.37 N-m (21 inch-pounds)
- It is recommended to use thread lock.
- If a washer and lock washer are used, the washer outer diameter must not be larger than the outer diameter of the bushing. A washer diameter of 10 mm is recommended if a washer is used.

## 10.2 Interfaces

### 10.2.1 CAN and UART

#### Contents

- *Ports*
- *CAN Messaging*
- *UART Messaging*

#### Ports

The OpenIMU300RI has two external ports; one UART port and one CAN bus port. Based on these available external ports, the OpenIMU300RI can be configured in several modes for communication with the external world.

The usage modes are:

<b>UART Mode</b>	<ul style="list-style-type: none"> <li>• Typically used during early development</li> <li>• Single UART for all messages, debug output, and firmware update</li> </ul>
<b>CAN + UART Mode</b>	<ul style="list-style-type: none"> <li>• Typically used during late development</li> <li>• Uses CAN Port for messages and firmware update</li> <li>• Single UART for all messages, debug output, and firmware update</li> </ul>
<b>CAN Mode</b>	<ul style="list-style-type: none"> <li>• Typically used for production</li> <li>• Uses CAN Port for messages and firmware update</li> </ul>

## CAN Messaging

To learn about CAN J1939 Messaging & Example Application For OpenIMU330RI, please see the following page:

1. CAN J1939 Messaging & Example Application

## UART Messaging

To learn more about the OpenIMU UART Messaging Framework, please see the following pages:

1. UART Messaging Framework

## 10.3 Pinout

### 10.3.1 OpenIMU300RI Connector Pinout

The OpenIMU300RI mating connector is the TE Connectivity 776531-1 (Ampseal-16 Housing “AS 16, 6P PLUG ASSY, RD, KEY 1”) or equivalent.

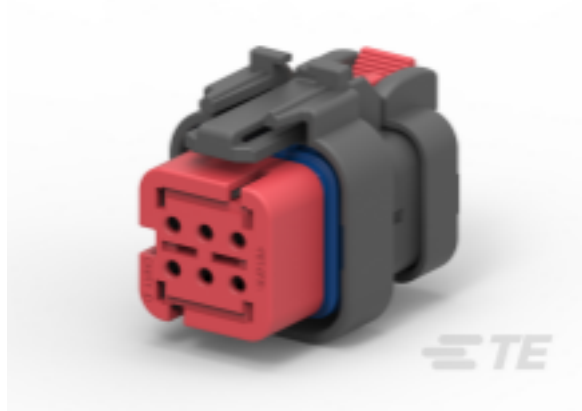


Fig. 3: OpenIMU300RI Connector

The pinout for that connector is shown in the following diagram. Pin 1 is in the upper right corner of the diagram.

The connector pin definitions are defined in the table below.

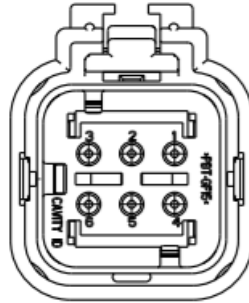


Fig. 4: OpenIMU300RI Connector Pinout

Pin	Signal
1	CAN H
2	CAN L
3	Ground
4	RS232 RX
5	RS232 TX
6	Power

Fig. 5: OpenIMU300RI Connector Pinout



**Note:** Power is applied to the OpenIMU300RI on pin 6. Pin 3 is ground. The OpenIMU300RI accepts an unregulated 4.9 to 32 VDC input. It is reverse polarity and ESD protected internally

---

### 10.3.2 ARM Cortex-M4 CPU

The OpenIMU300RI uses one of the powerful ST-Micro Cortex-M4 Microcontroller.

- FPU
- DSP instructions
- 1MByte Flash
- 192KB SRAM
- 168 MHz Clock
- Rich set of peripherals

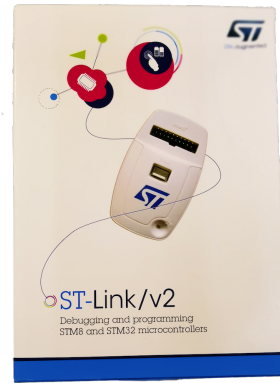
Learn more about the SoC used in the OpenIMU300RI at <http://www.st.com/en/microcontrollers/stm32f405rg.html>.

## 10.4 Eval Kit

### 10.4.1 OpenIMU300RI Eval Kit

The OpenIMU300RI evaluation kit includes:

- A robust and easy-to-use test fixture.
- An OpenIMU300RI IMU module attached to the test fixture with JTAG (SWD) 20-pin connector.
- An ST-LINK J-TAG debugger, a debugger cable, and a USB cable.
- A multiple-connector cable for RS232/CAN/Power connection.



### OpenIMU300RI Evaluation Kit Introduction

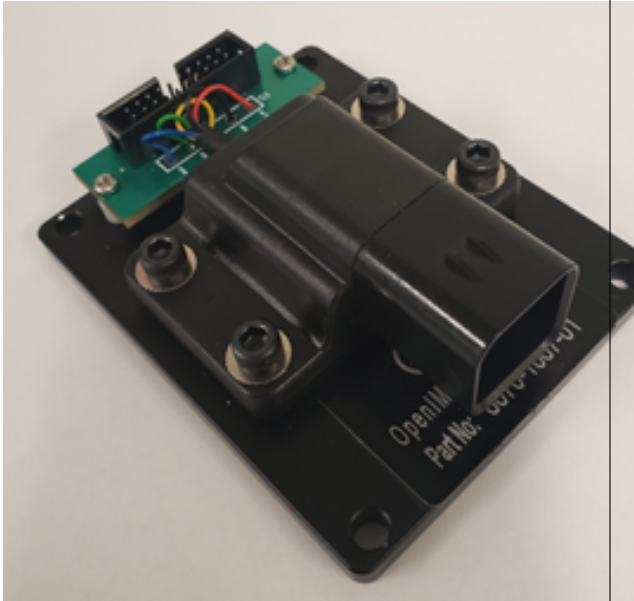

The OpenIMU evaluation kit is a hardware platform used to evaluate the OpenIMU300RI inertial navigation system and develop various applications based on this platform. It is supported by the Aceinna Navigation Studio, which provides easy access to the features of the OpenIMU300RI and explains how to integrate the device in a custom design. The Components section below provides the contents of the kit.

---

**Note:** An external DC power supply is required. The power supply must be able to provide 400mA at 4.9VDC to 32VDC.

The cable shown in the Evaluation Kit figure may look different than the cable that will be provided with the Evaluation Kit

---

	
<p><b>OpenIMU300RI Evaluation Unit</b> installed on test fixture with JTAG connector</p>	<p><b>OpenIMU300RI Evaluation Kit</b></p>

#### **OpenIMU300RI Evaluation Kit components**

##### **OpenIMU300RI unit**

OpenIMU300RI is 9 DOF (degrees of freedom) fully calibrated inertial unit. It is used as the base for development custom inertial navigation applications.

##### **OpenIMU300RI Evaluation Kit fixture and JTAG header board**

The OpenIMU300RI unit with JTAG header board are mounted on the test fixture. The JTAG

header provides means to debug/upload applications on evaluation unit.

### ST-Link debugger

The ST-Link debugger is a standard JTAG SWD debugger provided by STMicroelectronics company. It is used for in-system debugging/uploading of applications via SWD interface.

### OpenIMU300RI Breakout Cable

An included cable provides means of connecting unit to PC via RS232 interface, connecting unit to the CAN bus and powering up unit.

Next table provides connectors pin assignments in supplied cable

Signal Name	Unit Connector	RS232 Connector
GND	3	5
VIN	6	
RS232 TX	5	2
RS232 RX	4	3
CAN H	1	
CAN L	2	

## 10.4.2 OpenIMU300RI Evaluation Kit Setup

**To set up OpenIMU300RI evaluation kit you'll need to perform next steps:**

1. Install PC tools.
2. Unpack OpenIMU300RI evaluation kit.
3. Connect provided cable to OpenIMU300RI evaluation unit (see notes below).
4. Connect cable connector marked "RS232" to the PC serial port or to UCB-to-Serial adapter.

5. Connect cable connector marked “CAN” to the CAN bus or to the CAN traffic monitoring unit (like Vestor or Komodo or other).
6. Connect ST-Link debugger to the PC via USB cable. Make sure that ST-Link device appeared in “Device Manager”.
7. Connect 20-pin connector on OpenIMU300RI evaluation unit to ST-Link debugger using provided 20-pin flat cable.
8. Connect RED (+) and BLACK (GND) wires to external power supply (5 - 32V, 0.1A)
9. Turn ON power supply.

Now you are ready to debug and test your application.

- The following activities are addressed in the “Tools/Development Tools” section:
  - Download App with JTAG
  - Debugging with PlatformIO Debugger and JTAG Debug Adapter
  - Graphing & Logging IMU Data using the Acienna Navigation Studio

---

**Note:** The RS232/CAN/Power cable shown in the image is similar to the cable that will be supplied with the kit. It is for information only.

---

---

**Note:** The following directions are applicable for connecting cable to OpenIMU300RI evaluation unit:

- Align the keys on the unit and the cable connector.
  - Push the 6-pin cable connector into the unit connector until lock clicks.
  - If an extra lock required - push the red latch under the black latch. This prevents the disengagement button from being depressed.
-

**Note:** The following directions are applicable for disconnecting cable from OpenIMU300RI evaluation unit:

- If engaged, pull the red latch away from the connector toward the cable.
- Push down on the black disengagement button in the middle of the connector.
- Pull the cable connector away from the unit.

Next table provide connectors pin assignments in provided cable

Signal Name	Unit Connector	RS232 Connector	CAN Connector	Power Wires
GND	3	5		Black
VIN	6			Red
RS232 TX	5	2		
RS232 RX	4	3		
CAN H	1		7	
CAN L	2		2	

### OpenIMU300RI Connector

### OpenIMU Evaluation Kit Important Notice

This evaluation kit **is** intended **for** use **for** FURTHER ENGINEERING, DEVELOPMENT, DEMONSTRATION, OR EVALUATION PURPOSES ONLY. It **is not** a finished product **and** may **not** (yet) comply **with** some **or** any technical **or** legal requirements that are applicable to finished products, including, without limitation, directives regarding electromagnetic compatibility, recycling (WEEE), FCC, CE **or** UL (**except as** may be otherwise noted on the board/kit).  
(continues on next page)

→ Aceinna supplied this board/kit



(continued from previous page)

"AS IS," without  
↳any warranties, with  
↳all faults, at the  
↳buyer's and further  
↳users' sole risk. The  
user assumes all  
↳responsibility and  
↳liability for proper  
↳and safe handling  
↳of the goods. Further,  
the user indemnifies  
↳Aceinna from all  
↳claims arising from  
↳the handling or use  
↳of the goods. Due to  
the open  
↳construction of the  
↳product, it is the  
↳user's responsibility  
↳to take any  
↳and all appropriate  
precautions with regard  
↳to electrostatic  
↳discharge and  
↳any other technical  
↳or legal concerns.

(continues on next page)

(continued from previous page)

```

EXCEPT TO THE EXTENT
↳OF THE INDEMNITY SET
↳FORTH ABOVE, NEITHER
↳USER NOR ACEINNA
SHALL BE LIABLE
↳TO EACH OTHER FOR
↳ANY INDIRECT, SPECIAL,
↳ INCIDENTAL, OR
CONSEQUENTIAL DAMAGES.
No license
↳is granted under
↳any patent right
↳or other intellectual
↳property right
↳of Aceinna covering
or relating to any
↳machine, process, or
↳combination in which
↳such Aceinna products
↳or services might
be or are used.

```

### 10.4.3 OpenIMU300RI Eval Kit Fixture and Board

**Note:** The Power/CAN/RS232 cable shown is not the cable that will be provided in the kit. It is similar and is provided temporarily until an image of the actual cable is available.

The OpenIMU300RI module and the JTAG header board are mounted together on a precision fixture to assist in testing. The OpenIMU300RI Eval Kit provides interfaces to the main connector of the OpenIMU300RI and to the JTAG header board. The JTAG header, the OpenIMU300RI 9-pin D-sub connector, and the CAN 9-pin D-sub connector provide the means to connect the OpenIMU300RI Eval Kit to a PC.

#### OpenIMU300RI Default Coordinate System

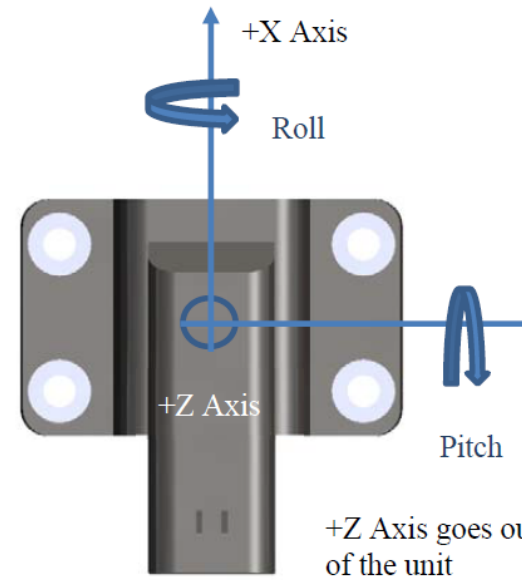
The OpenIMU default coordinate systems is shown below. In the reference IMU apps, a configuration setting is provided to control the coordinate system. These





Fig. 6: OpenIMU300RI Eval Kit Fixture and Board

configurable elements are known as **Configuration Parameters**.



## 10.5 Ready to Use Application

To learn about Ready-to-Use-Apps information & available for immediate download to your OpenIMU, please see the following pages:

1. [Ready-to-Use-Applications Information](#)
2. [Need to run the OpenIMU server before running one of the ready to use applications](#)
3. [Then upload a prebuilt app to your OpenIMU](#)

---

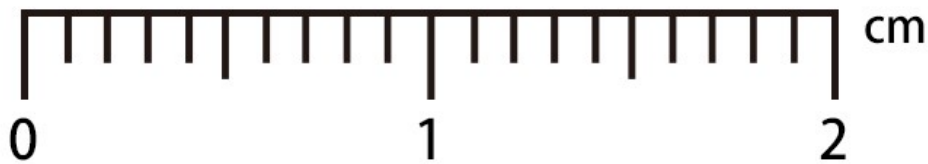
## OpenIMU330BI - Triple Redundant, 1.5 °/Hr, SMT Module

---

### Contents

- *Specifications*
- *SMT Process*
- *Interfaces*
- *Pinout*
- *Eval Kit*
- *Firmware Update*
- *Ready to Use Application*

The following image shows the OpenIMU330BI unit.



The OpenIMU330BI module integrates highly-reliable MEMS inertial sensors (acceleration, angular rate/gyro) in a miniature factory-calibrated package to provide consistent performance through the extreme operating environments.

It is easy to synchronize and interface with external GPS, as well as other sensors. The main feature of OpenIMU300BI is tripple redundancy for each inertial sensor.

- Integrated tripple redundant 3-Axis Angular Rate Sensors
- Integrated tripple redundant 3-Axis Accelerometer
- 80MHz STM32 M4 CPU with FPU
- SPI / UART Interfaces
- Sensors highest ODR is 200Hz
- Synchronization Input
- In-System Upgrade
- Ultra Small Size: 11x15x3 mm
- Wide Temp Range -40 to 85 ° C
- High Reliability > 50,000hr MTBF
- Low power

## 11.1 Specifications

### 11.1.1 OpenIMU330BI Environmental, Electrical, and Physical Specifications

#### ENVIRONMENT

Specifications	
Operating Temperature (°C)	-40 to 85
Non-Operating Temperature (°C)	-40 to 85
Enclosure	

#### ELECTRICAL

Specifications	
Input Voltage (VDC)	3.0 - 5.5
Power Consumption (mW)	< 250
Digital Interface	SPI or UART
Output Data Rate	up to 200Hz (SPI) up to 200Hz (UART)
Input Clock Sync	1KHz pulse (Configurable)

#### ABSOLUTE MAXIMUM RATINGS

Specifications	
Input Voltage (VDD)	3.0 to 5.5 V
Digital Input Voltage to GND	-0.3 to 3.6 V
Digital Output Voltage to GND	-0.3 to 3.6 V
Calibration Temperature Range	-40 to 85 C
Operating Temperature Range	-40 to 85 C
Non-Operating Temperature Range	-40 to 85 C

#### PHYSICAL

Specifications	
Size (mm)	11x15x3
Weight (gm)	1.0
Interface Connector	44 ball, BGA

#### VOLTAGE VALUES

Specifications	
Nominal voltage	3.3 V
All Pins Voltage	5 V
Reset Pin Max Voltage	3.6 V

#### VALUES

- Moisture Sensitivity Level (MSL) = 3
- The mechanical shock = 500 m/s<sup>2</sup>

**COMPLIANCE**

- OpenIMU330BI is RoHS and Reach Compliant.
- RoHS Compliant [download](#)
- Reach Compliant [download](#)

**INPUT VOLTAGE TOLERANCE**

No	MCU Pin	Name	Type	Description	Input Voltage Tolerance
A1		GND	P	Ground	
A2		GND	P	Ground	
A3		GND	P	Ground	
A4		GND	P	Ground	
A5		GND	P	Ground	
A6		GND	P	Ground	
A7		GND	P	Ground	
A8		GND	P	Ground	
B3		GND	P	Ground	
B4		GND	P	Ground	
B5		GND	P	Ground	
B6	PB11	DEBUG-RX	I	Receive debug data from user to IMU	5V
C2	PB10	DEBUG-TX	O	Transmit debug data from IMU to user	5V
C3		DNC	.	Not Used	
C6		GND	P	Ground	
C7		VDD	P	DC3.3V typical,input voltage range DC3.0V - 5.5V	
D3		GND	P	Ground	

Continued on next page

Table 1 – continued from previous page

D6		VDD	P	DC3.3V typical,input voltage range DC3.0V-5.5V	
E2		GND	P	Ground	
E3		VDD	P	DC3.3V typical,input voltage range DC3.0V - 5.5V	
E6		GND	P	Ground	
E7	PA11	GPIO1	I/O	GPIO	5V
F1	PA9	USER_UART_TX	O	Transmit IMU data to user	5V
F3	NRST	RST	I	Reset Signal In- put	3.6V
F6		GND	P	Ground	
F8		GND	P	Ground	
G2		GND	P	Ground	
G3	PB12	CS	I	SPI interface slave mode, CS signal	5V
G6	PB15	DIN	I	SPI interface slave mode, MOSI signal	5V
G7	PA10	USER_UART_RX	I	Receive commands from user to IMU	5V
H1		VDD	P	DC3.3V typical,input voltage range DC3.0V - 5.5V	

Continued on next page

Table 1 – continued from previous page

H3	PB14	DOUT	O	SPI interface slave mode, MISO signal	5V
H6	PB13	SCLK	I	SPI interface slave mode, Clock signal	5V
H8		GND	P	Ground	
J2	PA13	SWDIO	I/O	Data IO of SWD	5V
J3	PB3	PPS/SYNC	I	Sync signal from external device or 1PPS signal from GNSS module	5V
J4		VDD	P	DC3.3V typical,input voltage range DC3.0V - 5.5V	
J5		VDD	P	DC3.3V typical,input voltage range DC3.0V - 5.5V	
J6	PB5	DR	O	Data ready signal	5V
J7		GND	P	Ground	
K1	PH3	BOOT0	I	IMU boot mode control	5V
K3	PA14	SWCLK	I	Clock signal of SWD	5V
K6		VDD	P	DC3.3V typical,input voltage range DC3.0V - 5.5V	
K8	PA12	GPIO2	I/O	GPIO	5V

OpenIMU330BI Pin Voltage Tolerance doc download





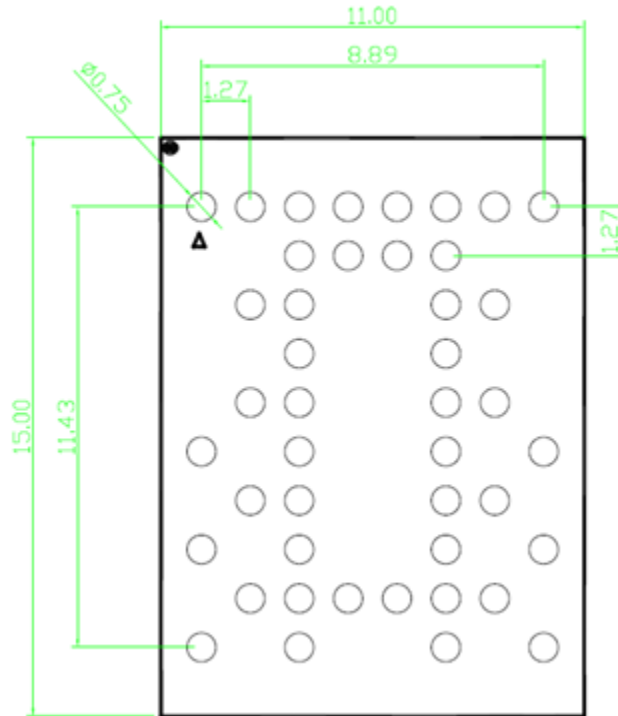


Fig. 1: **Figure 1: Recommended land pattern (unit: mm)**

## SOLDER REFLOW PROFILE

1. BGA ball material is SAC305.
2. The carrier board material of OpenIMU330BI is suggested Tg180 FR4.
3. Reflow profile for Pb free process
4. Reflow is limited by 2 times. Second reflow should be applied after device has cooled down to room temperature (25°C).
5. Recommended reflow profile for Pb free process is shown in Figure 2. The time duration of peak temperature (260°C) should be limited to 10 seconds.
6. Type 4 solder paste is recommended for a better SMT quality.
7. Use no clean flux to avoid product contaminated by cleaning solvent.
8. It is recommended use underfill glue to manage certain threats to the integrity of the solder joints of the OpenIMU330BI, including peeling stress and extended exposure to vibration. and underfill glue was not required that do not anticipate exposure to these types of mechanical stresses.

## PACKAGE OUTLINE DRAWING

Dimensions are in mm

OpenIMU330BI Land Pattern,Solder Reflow Profile and Package Outline doc download

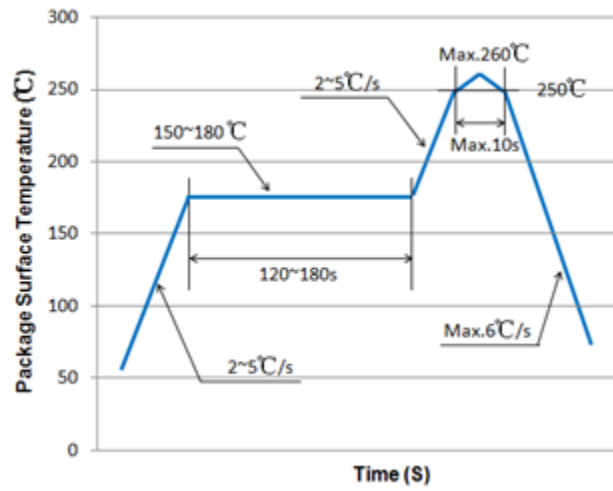


Fig. 2: Figure 2: Recommended solder reflow profile

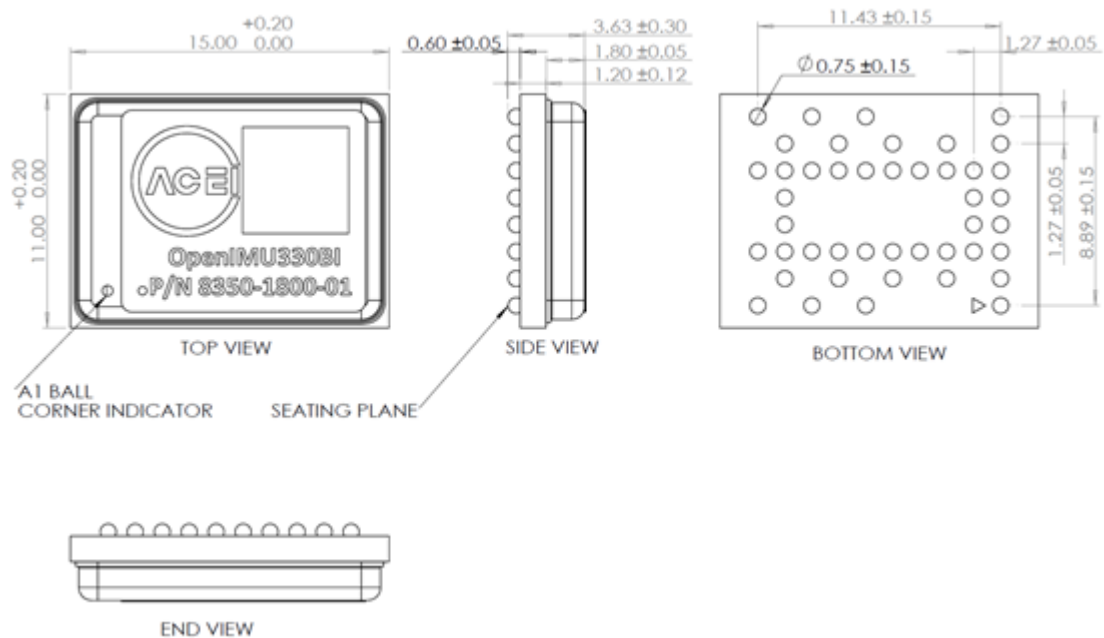


Fig. 3: Figure 3: Mechanical package outline dimensions

## 11.3 Interfaces

### 11.3.1 SPI and UART

#### Contents

- [Ports](#)
- [SPI & UART Messaging](#)

#### Ports

The OpenIMU300BI can be configured in a number of ways for communication with external world. There are two UART ports and one external SPI port.

Typical configurations include:

<b>2 UART Mode</b>	<ul style="list-style-type: none"> <li>• User UART</li> <li>• Debug UART</li> <li>• Debug UART</li> </ul>
<b>UART + SPI Mode</b>	<ul style="list-style-type: none"> <li>• User SPI Port</li> <li>• Debug UART</li> </ul>

#### SPI & UART Messaging

To learn more about the OpenIMU SPI & UART Messaging Framework, please see the following pages:

1. [SPI Messaging Framework](#)
2. [UART Messaging Framework](#)

## 11.4 Pinout

### 11.4.1 OpenIMU330BI Pinout and Function Descriptions

Dimensions are in mm

Schematic download

OpenIMU330BI pin assignment provided here - [download link](#).

### 11.4.2 ARM Cortex M4 CPU

The OpenIMU330BI uses ST's Cortex M4 series of Microcontrollers.

- FPU

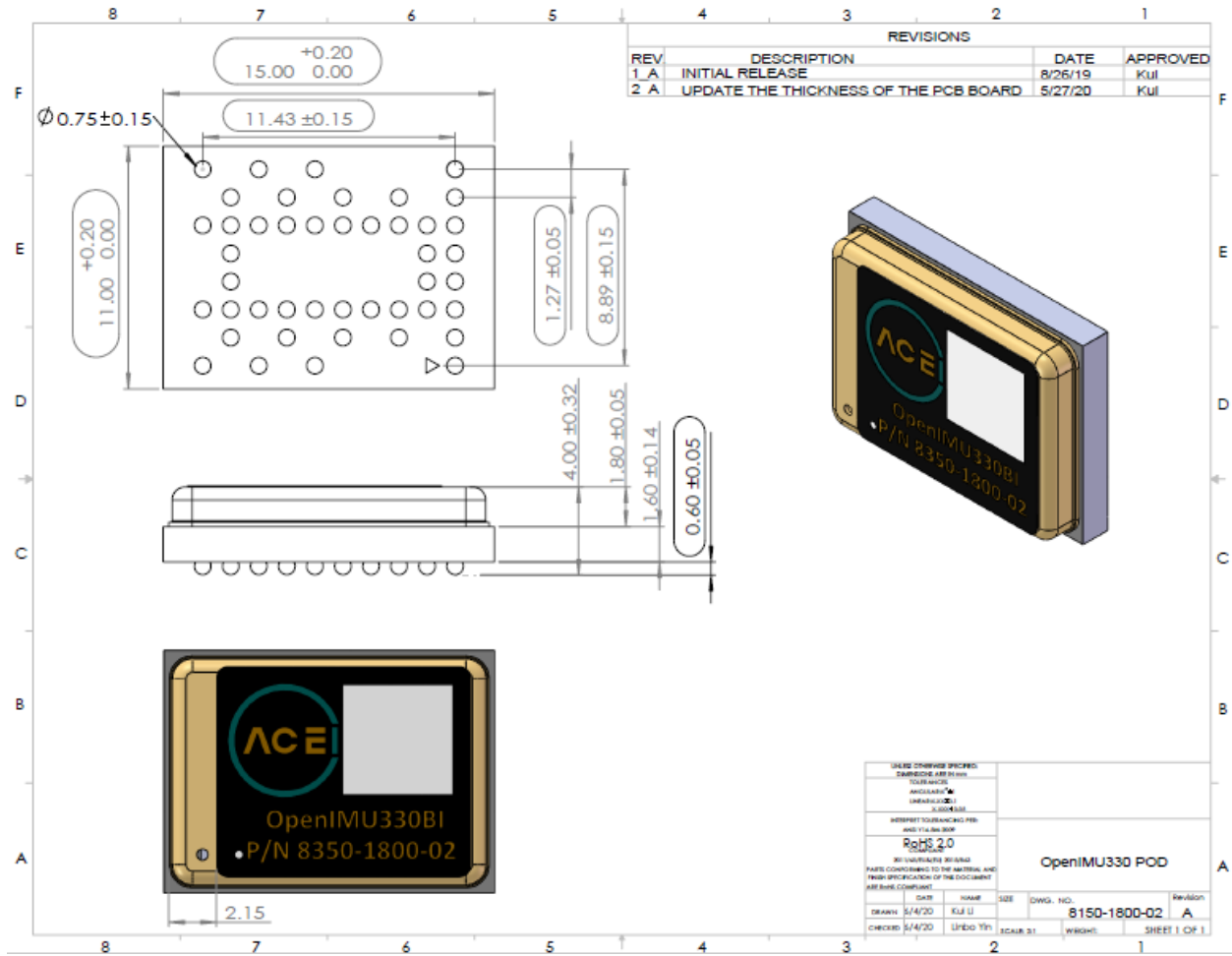


Fig. 4: OpenIMU330BI Module Mechanical Drawing

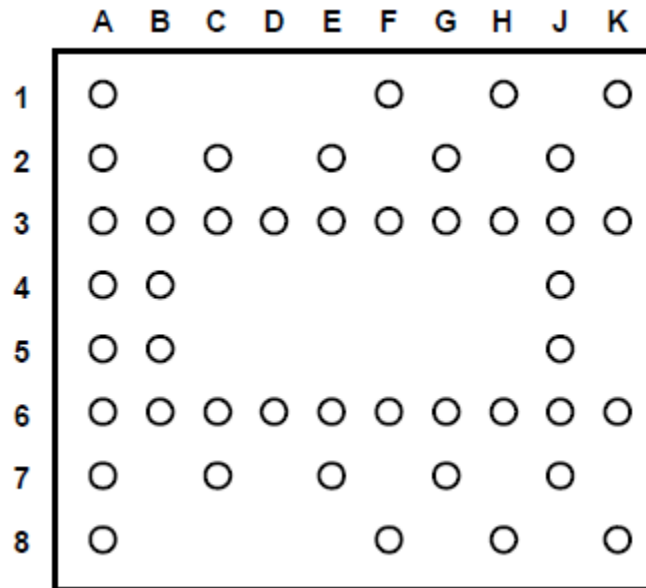
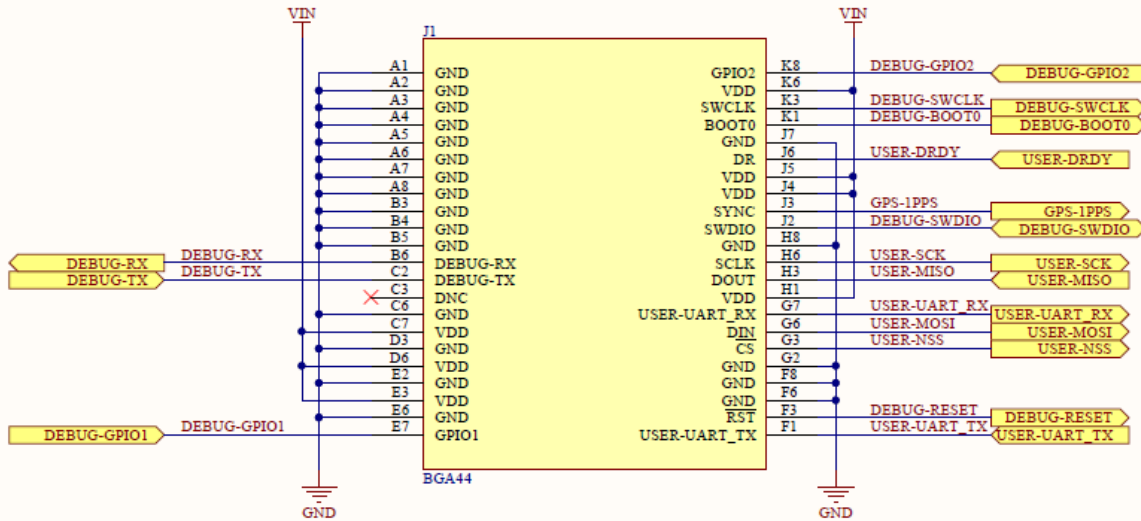


Fig. 5: Pin Assignments, Bottom View



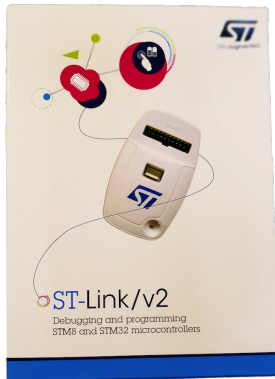
- DSP instructions
- 128KB Flash
- 64KB SRAM
- 80 MHz
- Rich Set of peripherals
- Low power

Learn more about OpenIMU330BI's CPU at <https://www.mouser.com/datasheet/2/389/stm32l431cb-956249.pdf>

## 11.5 Eval Kit

### 11.5.1 OpenIMU330BI Eval Kit

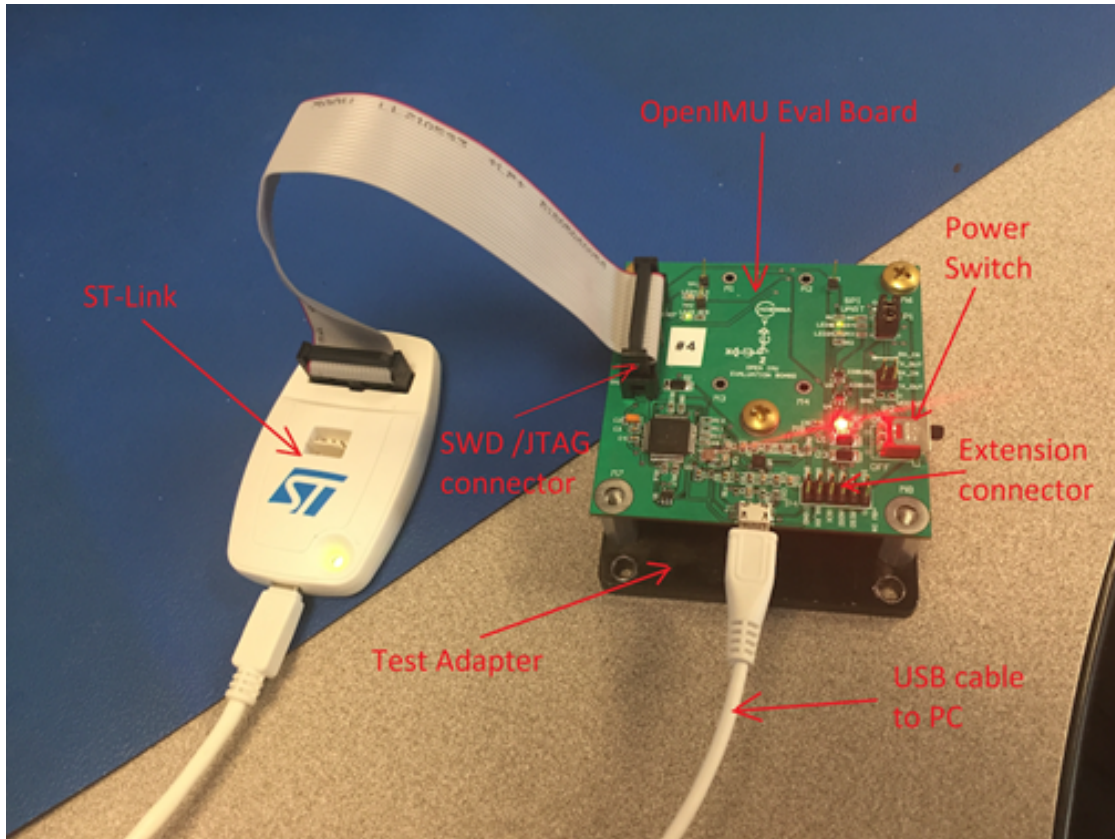
The OpenIMU330BI evaluation kit consists of a robust and easy-to-use eval board, a test fixture, the OpenIMU330BI IMU module, and an ST-LINK J-TAG pod.



## OpenIMU330BI Overview

### 1. Introduction

The OpenIMU evaluation kit is a hardware platform to evaluate the OpenIMU330BI inertial navigation system and develop various applications based on this platform. Supported by the Aceinna Navigation Studio the kit provides easy access to the features OpenIMU330BI and explains how to integrate the device in a custom design. The OpenIMU evaluation kit include OpenIMU330BI, evaluation board with various interface connectors and test adapter for mounting OpenIMU330BI unit.



## 2. Components

- OpenIMU Evaluation board, which includes:
  - Virtual COM-port USB interface, providing connectivity to OpenIMU330BI unit from PC
  - Connector for programming and debugging target via Serial Wire Debug (SWD) interface
  - Connector for interfacing OpenIMU330BI from custom-designed system.
  - Test terminals for connecting oscilloscope or logic analyzers to the dedicated OpenIMU330BI signals.
- OpenIMU330BI unit. Please note, that it installed on the bottom side of evaluation board.
- Test fixture adapter for convenient aligned mounting of OpenIMU evaluation board and OpenIMU330BI unit
- ST-Link debugger for in-system development of application code

### 2.1 OpenIMU330BI unit

OpenIMU330BI is 9 DOF (degrees of freedom) fully calibrated tripple redundant inertial unit. It is used as the base for development custom inertial navigation applications.

### 2.2 OpenIMU Evaluation board

OpenIMU Evaluation board designed to provide convenient way for communicating with OpenIMU330BI unit from PC, to expose serial and SPI interfaces to developer and to debug applications using ST-Link debugger vis SWD interface.

### 2.3 OpenIMU test adapter

OpenIMU test adapter used to firmly secure OpenIMU330BI unit and Open IMU evaluation board in precisely aligned position.



## 2.4 ST-Link debugger

St-Link debugger is standard debugger provided by STMicroelectronics company. It used for in-system debugging of applications via SWD interface.

## 3. Open IMU evaluation board Headers and Connectors

### 3.1 Connector for plugging in OpenIMU330BI unit (J2).

J2 is 20-pin connector and it used for connecting the OpenIMU330BI unit into Open IMU evaluation board. The pin functions are described in the table on the “OpenIMU Modules » OpenIMU330BI - EZ Embed Automotive Module » Connector Pinout - Including GPS Sensor Interface” page accessible from the Contents bar on the left.

### 3.2 Extension Header (P4)

OpenIMU evaluation board has 12-pin extension header. It designed to expose IMU interface signals to external system. The extension header pin functions described in table below

Pin	Main Function	Alternative Function
1	Power GND	Power GND
2	Power GND	Power GND
3	Serial Channel 1 RX (Input)	SPI Chip Select (SS) (Input)
4	IMU Data Ready (SPI interface Mode)	GPIO (UART interface mode)
5	User UART TX (Serial Channel 0) (Output)	SPI Clock (SCK) (Output)
6	Synchronization Input	1PPS Input from GPS
7		SPI Data (MOSI) (Input)
8	External Reset (NRST))	
9		SPI Data (MISO) (Output)
10	GPIO Output (IO2)	GPIO Input
11	Power VIN 5 VDC	Power VIN 5 VDC
12	GPIO Output (IO3)	GPIO Input
17	Debug UART TX	
19	Debug UART RX	

### 3.4 IMU interface type selection header (P1).

**Pins 1-2** define IMU **Interface Mode**:

If there is no connection between pins 1 and 2 (jumper is OFF) - **SPI** mode.

if there is connection between pins 1 and 2 (jumper is ON) - **UART** mode (default).

**In SPI mode:**

**Jumpers between pins 3-4 and 5-6 need to be taken OFF** to prevent interference between SPI bus signals (SS and MISO) and serial interface signals from FTDI chip.  
IMU SPI interface signals (MISO, MOSI, SS, SCK, DRDY) routed to header P4.

---

**Note:** On SPI interface IMU acts as a **SLAVE** device.

---



---

**Note:** Not all provided application examples support SPI interface mode. Please refer to specific example for details.

---

**In UART mode:**

Jumper between pins **3-4** should be **“ON”** (default) if IMU **Serial Channel 0** ( USER main channel ) needs to be routed to PC via USB connection (on first in the row enumerated USB virtual COM port. See p.6).

Jumper between pins **3-4** should be **OFF** if IMU **Serial Channel 0** needs to be accessed from P2 connector.

**3.5 IMU Serial Debug Channel mode selection header (P2).**

Jumpers between pins **1-2** and **3-4** should be **ON** if IMU **Debug Serial** needs to be routed to PC via USB connection, for example in case of using IMU Debug Serial Channel for streaming out debug information to PC or as CLI interface (on third in the row enumerated USB virtual COM port. See p.6).

Jumpers between pins **1-2** and **3-4** should be **OFF** if IMU **Debug Serial Channel** needs to be routed to some external device (for example GPS). In this case **pin 2 is RX** (to IMU) and **pin 4 is TX** (from IMU).

**3.6 SWD (JTAG) connector (P3).**

20-pin connector P3 used for connecting ST-Link or J-Link debuggers to the IMU for in-system debugging of applications via SWD interface. It has standard pin-out.

Pin	Main Function
1, 2	Vref
4, 6, 8, 10 , 12, 14, 16, 18, 20	GND
7	SWDIO
9	SWCLK
15	nRST
19	3.3V from debugger

**3.7 USB connector (J3)**

USB connector used for powering up the IMU and evaluation board. Also its used to providing connectivity from PC to IMU via virtual serial ports. Up to 3 exposed IMU serial interfaces can be routed to PC.

#### 4. OpenIMU evaluation board LED indicators

Evaluation board has few LED indicators for visual monitoring of data traffic on serial ports:

**LED2** indicator reflects activity on RX line of IMU main (user) serial interface (traffic to IMU)

**LED1** indicator reflects activity on TX line of IMU main (user) serial interface (traffic from IMU)

**LED3** indicator while lit indicates presence of the power (in case switch SW1 is “ON”)

**LED4** indicator reflects activity on GPIO3 (lit if high)

**LED5** indicator reflects activity on GPIO2 (lit if high)

#### 5. Open IMU evaluation board power

Power to OpenIMU evaluation board provided by USB. To power system up - connect USB cable to connector J1 and turn “ON” switch SW1.

#### 6. Communication with IMU from PC

The OpenIMU evaluation board has an FTDI chip FT4232 installed. This chip provides 4 virtual serial ports. When evaluation board set up to force IMU interface in UART mode (see p.3.4) up to 3 serial ports on IMU can communicate with PC. When evaluation board connected to PC and power switch turned “ON” in Device Manager board will appear as **4 new consecutive virtual COM ports**.

First in a row virtual port is routed to IMU’s main UART channel (Serial channel 0) (pins 3 and 4 on J2), and usually dedicated for sending commands to IMU and capturing responses and periodic messages from IMU. It usually used by python driver to establish communication between IMU and Aceinna Navigation Studio.

Third in a row virtual port routed to IMU’s Debug Serial Channel (pins 17 and 19 on J2) and usually used as a debug/CLI serial channel .

### 11.5.2 OpenIMU330BI Evaluation Kit Setup

To set up OpenIMU330BI evaluation kit for development you’ll need to perform next steps:

1. Unpack OpenIMU330BI evaluation kit.
2. Push power switch to “OFF” position.
3. Connect OpenIMU330BI evaluation board to the PC via USB cable. USB connection provides power to the test setup as well as connectivity between PC and IMU serial ports.
4. Connect ST-Link debugger to the PC via USB cable.
5. Connect OpenIMU330BI evaluation board to ST-Link debugger using provided 20-pin flat cable.
6. Push power switch to “ON” position.

Now you are ready to debug and test your application.

- The following activities are addressed in the “Development Tools” section:
  - Download App with JTAG
  - Debugging with PlatformIO Debugger and JTAG Debug Adapter

– Graphing & Logging IMU Data using the Acienna Navigation Studio

### OpenIMU Evaluation Kit Important Notice

This evaluation kit **is** intended **for** use **for** FURTHER ENGINEERING, DEVELOPMENT, DEMONSTRATION, OR EVALUATION PURPOSES ONLY. It **is not** a finished product **and** may **not** (yet) comply **with** some **or any** technical **or** legal requirements that are applicable to finished products, including, without limitation, directives regarding electromagnetic compatibility, recycling (WEEE), FCC, CE **or** UL (**except as** may be otherwise noted on the board/kit). Aceinna supplied this board/kit "AS IS," without **any** warranties, **with** all faults, at the buyer's **and further users'** sole risk. The user assumes **all** responsibility **and** liability **for** proper **and** safe handling of the goods. Further, the user indemnifies Aceinna **from all** claims arising **from the** handling **or** use of the goods. Due to the **open** construction of the product, it **is** the user's **responsibility to take any and all appropriate** precautions **with** regard to electrostatic discharge **and any** other technical **or** legal concerns.

EXCEPT TO THE EXTENT OF THE INDEMNITY SET FORTH ABOVE, NEITHER USER NOR ACEINNA SHALL BE LIABLE TO EACH OTHER FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES.

No license **is** granted under **any** patent right **or** other intellectual **property** right of Aceinna covering **or** relating to **any** machine, process, **or** combination **in** which such Aceinna products **or** services might be **or** are used.

**Note:** In OpenIMU330BI EVK by default signs of readings on X and Z axes are flipped in comparison to the Coordinate Frame drawing on top of the EVK board since OpenIMU330BI unit mounted upside-down in the EVK.

## 11.5.3 OpenIMU330BI Eval Board & Coordinate Frame

### OpenIMU330BI Eval Board and Fixture

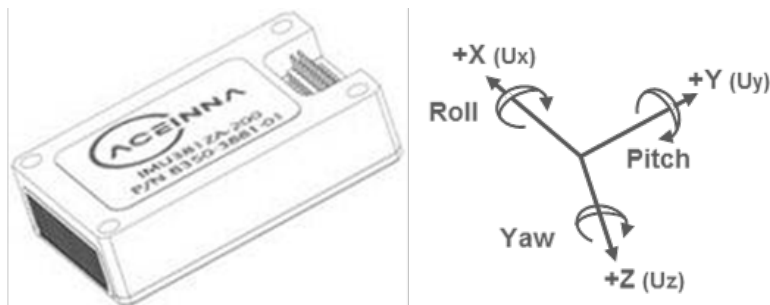


The OpenIMU330BI Eval Board is attached to a fixture for easy handling and isolation of the back side of the board

from any contact. The EVB interfaces to the main connector of the OpenIMU330 evaluation module. The EVB and IMU module are mounted together to a precision fixture to assist in testing. The OpenIMU330EVB uses an FTDI 4-port Serial-to-USB converter to allow you to communicate with between the OpenIMU serial ports and a laptop computer. There are also jumper connections to use to connect to the device's primary SPI port. Use the JTAG interface to directly download compiled code to the device quickly.

### OpenIMU330BI Eval Module Default Coordinate System

The OpenIMU default coordinate systems is shown below. In the reference IMU apps, a configuration setting is provided to control the coordinate system. These configurable elements are known as **Configuration Parameters**.



## 11.6 Firmware Update

### 11.6.1 OpenIMU330BI Firmware Update

In comparison to OpenIMU300ZI and OpenIMU300RI, OpenIMU330BI does not have built-in bootloader. The reason behind it that OpenIMU300BI uses processor with less resources and sometimes if application is big – there would be now room to fit it in if bootloader present. There are two scenarios how FW update can be performed for OpenIMU330BI.

#### Using JTAG (SWD) interface

OpenIMU330BI has standard SWD interface. Next pins are used:

Pin #	Pin Function	Note
K3	SWCLK	
J2	SWDIO	
F3	RESET	
	VIN	Reference voltage
	GND	Ground

SWD interface allows to perform programming of application into unit from development environment or using special utilities, for example ST-Link Utility.

#### Note:

- Application image should be programmed from address 0x08000000
- Last 6 sectors of MCU flash should not be erased during programming. They contain calibration parameters.

- For unit to work properly last 6 sectors need to be write-protected. Write protection performed at the factory but in case of unit recovery it needs to be performed again.
- To be able to recover unit read and save full original image first (from address 0x08000000, length 0x20000).

### Using built-in MCU bootloader

Application image can be programmed into the unit via serial interface using built into the processor boot loading capability.

Next pins on OpenIMU330BI are used in this case:

Pin #	Pin Function	Notes
K1	BOOT0	3.3 V
G7	USER_UART_RX (to unit)	0 – 3.3V
F1	USER_USRT_TX (from unit)	0 – 3.3V
F3	RESET	0 – 3.3V Optional
	GND	

Next sequence needs to be executed to force unit into boot loading mode:

1. Connect serial RS232 interface from PC to unit using RS232-TTL convertor. There may be also direct USB-TTL serial adapter.
2. Provide HIGH level on BOOT0 pin.
3. Power up unit or apply RESET signal (active low. Time > 10 milliseconds).
4. Start custom boot loading utility or ST Micro utility and follow the steps in the documentation.

User can choose to implement their own boot loader or use utilities provided by ST-Micro.

In first case find boot loading application note AN3155 here:

[https://www.st.com/content/st\\_com/en/search.html?q=AN3155-t=resources-page=1](https://www.st.com/content/st_com/en/search.html?q=AN3155-t=resources-page=1)

In second case find ST Flash Loader Utility here:

<https://www.st.com/en/development-tools/flasher-stm32.html>

#### Note:

- In case if unit is not recognized by ST Flash Loader, place file STM32L4\_128K.STmap (download below) into MAP directory (created during tool installation)
- Application image should be programmed from address 0x08000000
- Last 6 sectors of MCU flash should not be erased during programming. They contain calibration parameters.
- For unit to work properly last 6 sectors need to be write-protected. Write protection performed at the factory but in case of unit recovery it needs to be performed again.
- To be able to recover unit read and save full original image first (from address 0x08000000, length 0x20000).

File STM32L4\_128K.STmap [download](#)

## 11.7 Ready to Use Application

To learn about Ready-to-Use-Apps information & available for immediate download to your OpenIMU, please see the following pages:

1. [Ready-to-Use-Applications Information](#)
2. [Need to run the OpenIMU server before running one of the ready to use applications](#)
3. [Then upload a prebuilt app to your OpenIMU](#)



---

## OpenIMU335RI - Triple-Redundant Rugged Industrial CAN Module

---

### Contents

- *Introduction*
- *OpenIMU335 Product Page*
- *Datasheet*
- *User Manual*
- *Eval Kit*
- *OpenIMU Development Environment*
- *Ready to Use Applications*

## 12.1 Introduction

The ACEINNA OpenIMU335RI is an easy-to-use high-performance 6-DOF (9 DOF is optional) open inertial platform packaged in a rugged sealed over-molded plastic housing. The OpenIMU335RI includes triple-redundant 3-Axis MEMS accelerometers and rate gyros which are fully calibrated over the operating temperature range. A 3-axis magnetic sensor is also available as an option. The processing power is provided by a 168MHz ARM M4 CPU with a Floating Point Unit. The OpenIMU335RI runs the OpenIMU open-source stack that includes an optimized 16-state Kalman Filter for Attitude and GPS-Aided PositionVelocity-Time (PVT) measurement. A free tool-chain based on VS Code supports PC, MAC, and Ubuntu.

- Hardware
  - 0.1 degrees of accuracy over temperature and angle
  - Precision 3-axis MEMS Accelerometer
  - Low-Drift 3-axis MEMS angular rate sensor



Fig. 1: OpenIMU335 Module

- Triple-redundant architecture with fault detection
  - 3-axis AMR Magnetometer (Optional)
  - CAN 2.0 and RS232 Interfaces
  - 168 MHz ARM M4
  - Wide Temp Range, -40C to +85C
  - Wide Supply Voltage Range, 9 V – 32 V
  - IP67 Ampseal Connector
  - High Reliability, MTBF > 50k hour
- 
- Firmware and Firmware Support
    - In-System Firmware Upgrade
    - Open Source Tool Chain
    - Open Source Algorithms (VG / AHRS / INS)
    - Built in 16-State Open Source Extended State Kalman Filter
    - Open Community & Support
    - Aceinna Navigation Studio Interface

The default coordinate frame for the OpenIMU335RI is given in the below image. The coordinate frame can be changed via RS-232 or CAN message. Refer to the user manual of the part for details.

## 12.2 OpenIMU335 Product Page

The OpenIMU335RI product page is at: <https://www.aceinna.com/inertial-systems/OpenIMU335RI>. Refer to this page for a product summary and direct links to the datasheet, user manual and DBC file for the CAN interface.

## 12.3 Datasheet

The datasheet of the OpenIMU335RI can be downloaded from this [datasheet link](#). Refer to this document for:

- Detailed Specifications
  - Performance
  - Electrical
  - Physical
  - Environmental
  - EMC
- Features
- Qualification Plan Summary
- Module Dimensions

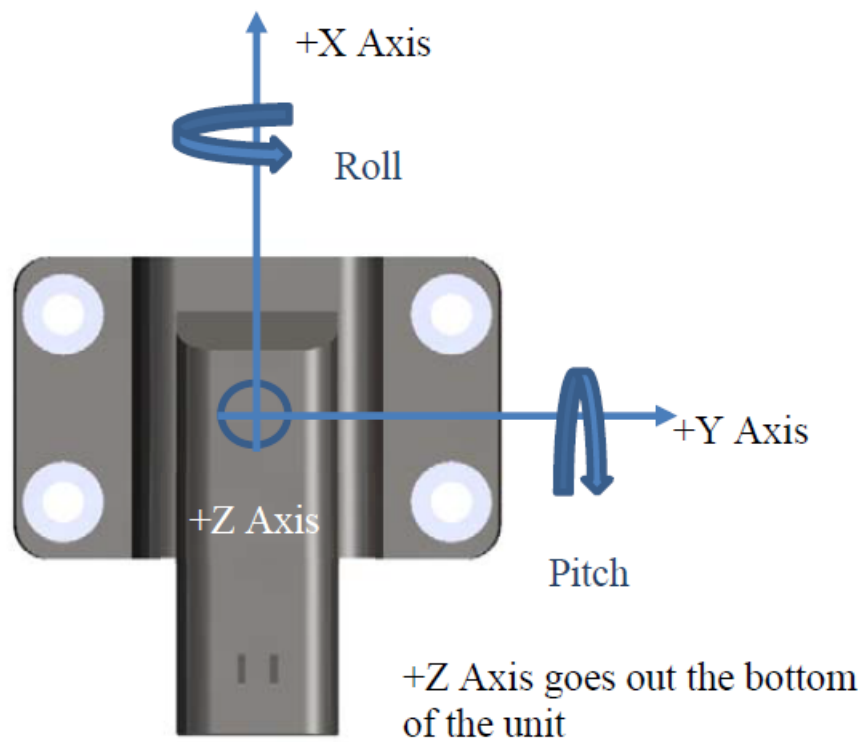


Fig. 2: OpenIMU335RI Default Coordinate Frame

- Part Ordering Information

## 12.4 User Manual

The user manual for the OpenIMU335RI can be downloaded from this [user manual link](#). Refer to this document for details of the:

- Electrical and Mechanical Interfaces
- Theory of Operation
- Safety Features
- CAN Interface and CAN Messages
- RS-232 Interface and RS-232 Messages
- Bootloader
- Warranty and Support Information

## 12.5 Eval Kit

### 12.5.1 OpenIMU335RI Eval Kit

#### OpenIMU335RI Evaluation Kit Introduction

The OpenIMU evaluation kit is a hardware platform used to evaluate the OpenIMU335RI inertial navigation system and develop various applications based on this platform. A difference to standard part is that a JTAG (SWD) 20-pin header is brought out for code development. It is supported by Aceinna Navigation Studio, which enables the user to quickly evaluate the part. The Components section below details the contents of the kit.

---

**Note:** An external DC power supply is required. The power supply must be able to provide 100mA at 9VDC to 32VDC.

---



### OpenIMU335RI Evaluation Kit components

OpenIMU335RI Evaluation Kit fixture and JTAG header board

- The OpenIMU335RI unit with JTAG header board are mounted on the test fixture. The JTAG header provides a means to debug/upload applications to the evaluation unit.



#### ST-Link debugger

- The ST-Link V2 programmer / debugger is a standard JTAG SWD debugger provided by STMicroelectronics company. It is used for in-system debugging/uploading of applications via SWD interface.



### OpenIMU335RI Breakout Cable

- An included cable provides a means of connecting the unit to a PC via RS232 interface, connecting the unit to the CAN bus, and powering the unit. The next table shows the connector pin assignments of the supplied cable.

Signal Name	Unit Connector	RS232 Connector	CAN Connector	Power Wires
GND	3	5		Black
VIN	6			Red
RS232 TX	5	2		
RS232 RX	4	3		
CAN H	1		7	
CAN L	2		2	

### OpenIMU335RI Connector

The connector of the breakout cable is shown in the following image. See the notes below for details of how to connect to and disconnect from the OpenIMU335RI.

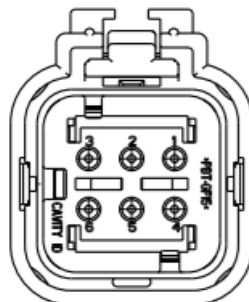
The pin numbers are as follows:

---

**Note:** To connect the cable to the OpenIMU300RI evaluation unit:

- Align the keys on the unit and the cable connector.
- Push the 6-pin cable connector into the unit connector until lock clicks.





- If an extra lock is required - push the red latch under the black latch. This prevents the disengagement button from being depressed.

**Note:** To disconnect the cable from the from OpenIMU335RI evaluation unit:

- If engaged, pull the red latch away from the connector toward the cable.
- Push down on the black disengagement button in the middle of the connector.
- Pull the cable connector away from the unit.

## 12.5.2 OpenIMU335RI Evaluation Kit Setup

To get started with the OpenIMU335RI evaluation kit connect the breakout cable to the evaluation kit.

- Connect the RS232 connector of the cable to a PC if you wish to evaluate using Aceinna Navigation Studio.
- To evaluate the part using the CAN interface simply connect to either a CAN analyzer, or network, and refer to the CAN Port Interface Definition section of the [user manual](#).
- Connect RED (+) and BLACK (GND) wires to an external power supply (9 - 32V, 0.1A)

Refer to the [Aceinna Navigation Studio website](#) where there is documentation on how to:

- Download a PC server application that will allow you to evaluate the part over the RS-232 interface using the Chrome® web browser: <https://developers.aceinna.com/devices/connect>
- Update the firmware on the OpenIMU335RI using one of Aceinna's pre-compiled applications: <https://developers.aceinna.com/code/apps>
- Install the OpenIMU programming environment for user code development: <https://developers.aceinna.com/docs/install>

The following activities are addressed in the [Tools](#) section:

- How to upload an App via JTAG
- Debugging with the PlatformIO Debugger and JTAG Debug Adapter
- Graphing & Logging IMU Data using the Aceinna Navigation Studio

### OpenIMU Evaluation Kit Important Notice

This evaluation kit **is** intended **for** use **for** FURTHER ENGINEERING, DEVELOPMENT, DEMONSTRATION, OR EVALUATION PURPOSES ONLY. It **is not** a finished product **and** may **not** (yet) comply **with** some **or** any technical **or** legal requirements that are applicable to finished products, including, without limitation, directives regarding electromagnetic compatibility, recycling (WEEE), FCC, CE **or** UL (**except as** may be otherwise noted on the board/kit). Aceinna supplied this board/kit "AS IS," without any warranties, **with** all faults, at the buyer's and further users' sole risk. The user assumes all responsibility **and** liability **for** proper **and** safe handling of the goods. Further, the user indemnifies Aceinna **from** all claims arising **from** the handling **or** use of the goods. Due to the open construction of the product, it **is** the user's responsibility to take any and all appropriate

(continues on next page)

(continued from previous page)

precautions **with** regard to electrostatic discharge **and** **any** other technical **or** legal ↵  
↵concerns.

EXCEPT TO THE EXTENT OF THE INDEMNITY SET FORTH ABOVE, NEITHER USER NOR ACEINNA  
SHALL BE LIABLE TO EACH OTHER FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR  
CONSEQUENTIAL DAMAGES.

No license **is** granted under **any** patent right **or** other intellectual **property** right of ↵  
↵Aceinna covering

**or** relating to **any** machine, process, **or** combination **in** which such Aceinna products **or** ↵  
↵services might

be **or** are used.

## 12.6 OpenIMU Development Environment

For details of the OpenIMU development environment refer to:

## 12.7 Ready to Use Applications

To learn about Ready-to-Use-Apps available for immediate download to your OpenIMU, please see the following pages:

1. [Information about ready-to-Use-Applications](#)
2. [Run the OpenIMU server before uploading or using a prebuilt app](#)
3. [Upload a prebuilt app to your OpenIMU](#)

## **Part III**

# **Dev Support Algorithms**

---

## OpenIMU Hardware/Software Interface

---

This section describes firmware-configurable connections from external hardware to the OpenIMU platform. In particular, it describes how external connections are connected and how the platform code modules are inputs and work together. If the platform cannot support the connection (such as the OpenIMU300RI) then this section is not applicable. Examples of external sources include:

- Synchronization to external clock signals
- GPS receiver input
- Odometer input

If the input is provided via a software interface, such as CAN, then this section is not applicable.

### 13.1 Synchronization to External Clock Signals

#### Contents

- *Connecting an External Clock*
- *Configuration Settings*
- *Post-Synchronization Operations*

External clock signals provide a way for the system to synchronize sensor sampling and processing, as well as algorithm operations, to an external source. In general, this can improve the performance of inertial systems and algorithms by enhancing the time-relevancy measurement of the sensor output, via a highly accurate timestamp with micro-second resolution, or by enabling algorithm updates with information that is more timely.

By default the system is configured to synchronize to a 1 kHz signal; all that is required is to connect the signal to the OpenIMU device. However, to enable synchronization lock to an external 1 Hz signal (such as the GPS PPS signal) the user must configure the firmware to operate with a 1 Hz external clock by calling `platformEnableGpsPps(TRUE)`; during system initialization.

### 13.1.1 Connecting an External Clock

To synchronize the system to an external clock, the first step is to connect the signal to the appropriate pin for the device being used:

1. OpenIMU300ZI: Pin 2 serves as the clock input ([OpenIMU300ZI Connector Pinout](#))
2. OpenIMU330BI: Pin J3 serves as the clock input ([OpenIMU330BI Connector Pinout](#))
3. OpenIMU300RI: No pin available for synchronization to external clock

### 13.1.2 Configuration Settings

The second step in synchronizing to an external clock signal is to configure the firmware to use the external signal connected to the external clock pint. As mentioned above, the system is automatically configured to use a 1 kHz. To connect and synchronize to a 1 Hz signal, the user must configure the firmware by calling *platformEnableGpsPps(TRUE)*;. An ideal place to perform this task is during initialization of the user-generated algorithm, commonly executed in *dataProcessingAndPresentation.c*.

The following code snippet shows how the INS application initializes the system to synchronize to the 1 PPS signal:

```
// Next function is common for all platforms, but implementation of the methods
// inside is
// platform-dependent. Call to this function is made from DataAcquisitionTask during
// the
// initialization phase. All user algorithm and data structures should be initialized
// here, if used.
void initUserDataProcessingEngine()
{
    InitUserAlgorithm();           // default implementation located in file user_
    // algorithm.c
    platformEnableGpsPps(TRUE);    // Init PPS sync engine
}
```

### 13.1.3 Post-Synchronization Operations

Once the system is configured to synchronize to the expected input clock signal, there are a variety of functions available to take advantage of the synchronization. Two such functions are:

1. *platformGetSolutionTstampAsDouble()*
2. *platformGetEstimatedITOW()*

Additionally, users can take advantage of the PPS synchronization by monitoring for, and responding to, the PPS signal using the function *platformGetPpsFlag(TRUE)*;, where TRUE indicates that the PPS flag is reset after reading.

The time delay between the PPS signal and the availability of the GPS data can be measured using the function *getSystemTime()*. This measurement can then be used to account for by the delay in the algorithm.

## 13.2 Connecting to a GPS Receiver Input Signal

#### Contents

- [Connecting a GPS Receiver](#)

- *Configuration Settings*
- *Post-Connection Operations*

## THIS IS A PLACEHOLDER PAGE

External GPS receiver signals provide a way for the system to acquire knowledge of the system position and velocity in the Earth's reference frame. This page serves as a placeholder for the description of the connection to the GPS receiver. In addition to connecting to the receiver, the unit needs to know what format the input signal will be (for example, NMEA or NovAtel) and what the message will be (GGA vs VTG, etc.).

### 13.2.1 Connecting a GPS Receiver

To connect the system to an external GPS receiver, the first step is to connect the TX and RX lines to the appropriate pins for the device being used:

1. OpenIMU300ZI: Pins 17 and 19 serve as the GPS input pins ([OpenIMU300ZI Connector Pinout](#))
2. OpenIMU330BI: Pin X and Y serve as the GPS input pins ([OpenIMU330BI Connector Pinout](#))
3. OpenIMU300RI: No pin available for connection to GPS receiver

### 13.2.2 Configuration Settings

The second step in connecting to a GPS receiver is to configure the firmware to use the signals. An ideal place to perform this task is during initialization of the user-generated algorithm, commonly executed in *dataProcessingAndPresentation.c*.

### 13.2.3 Post-Connection Operations

Once the system is configured to accept and decode the GPS receiver input, ...

1. *Function1()*
2. *Function2()*

## 13.3 Connecting to an External Odometer

### Contents

- *Connecting an Odometer*
- *Configuration Settings*
- *Post-Connection Operations*

## THIS IS A PLACEHOLDER PAGE

External odometer signals provide a way for the system to acquire knowledge of the system velocity and heading relative to the vehicle-frame. This page serves as a placeholder for the description of the connection to the odometer. In addition to connecting to the odometer, the unit needs to know what format the input signal will be and what the message will be.

### 13.3.1 Connecting an Odometer

To connect the system to a non CAN-based, external odometer, the first step is to connect the signal lines to the appropriate pins for the device being used:

1. OpenIMU300ZI: Pins X and Y serve as the input pins ([OpenIMU300ZI Connector Pinout](#))
2. OpenIMU330BI: Pin X and Y serve as the input pins ([OpenIMU330BI Connector Pinout](#))
3. OpenIMU300RI: No pin available for connection to odometer

### 13.3.2 Configuration Settings

The second step in connecting to an odometer is to configure the firmware to use the signals. An ideal place to perform this task is during initialization of the user-generated algorithm, commonly executed in *dataProcessingAndPresentation.c*.

### 13.3.3 Post-Connection Operations

Once the system is configured to accept and decode the odometer input, ...

1. *Function1()*
2. *Function2()*



This section develops the equations that form the basis of an Extended Kalman Filter (EKF), which calculates position, velocity, and orientation of a body in space<sup>1</sup>. In a VG, AHRS, or INS<sup>2</sup> application, inertial sensor readings are used to form high data-rate (DR) estimates of the system states while less frequent or noisier measurements (GPS and inertial sensors) act as references to correct errors in the system.

In addition to deriving the EKF equations, this description presents a measurement model based on Euler angles, which result from accelerometers, magnetometers, and GPS readings. Following that it describes implementations that result in improved solutions under both static and dynamic conditions. Finally, a series of examples illustrate existing VG, AHRS, and INS algorithms.

The algorithm development description is broken up into a series of sections that build upon one another, as follows:

- Coordinate Frames
- Attitude Parameters
- Sensors
- Extended Kalman Filter
- Process Models
- Measurement Model
- Measurement Vector
- Innovation (Measurement Error)
- Magnetic Alignment
- References

---

<sup>1</sup> This discussion presupposes a certain amount of knowledge. Details related to differential equations, linear algebra, multi-variable calculus, stochastic processes, etc. are not described.

<sup>2</sup> A VG uses rate-sensors and accelerometers to estimate roll and pitch. An AHRS incorporates magnetometer readings to the VG to estimate heading. An INS adds GPS messages to the VG or AHRS to estimate position and velocity or provide a way to estimate heading without magnetometers.

## 14.1 Coordinate Frames

A body's position and orientation can only be measured relative to another set of basis vectors (coordinate-frame). In this formulation, inertial sensors provide the information to compute the attitude and position of a body in space relative to an "inertial" frame, such as the Earth-Centered, Earth-Fixed frame (ECEF) or the North/East/Down-frame (NED)<sup>1</sup>. The equations to come use the superscripts listed in [Table 1](#) to specify the frame in which a variable is measured.

Table 1: **Table 1: Frames and their Identifiers used throughout Algorithm Derivation**

Frame	Superscript	Description
ECEF-Frame	E	Frame aligned with Earth's axis (z-axis parallel to axis-of-rotation, x-axis exits at the equator through the prime-meridian); rotates with the Earth ( <b>not shown in Figure 1</b> )
NED-Frame	N	Frame aligned with the local tangent-frame (z-axis parallel to the gravity vector) with the x-axis aligned with true or magnetic north. Red lines in <i>Figure 1</i> .
Perp-Frame	$\perp$	Frame aligned with the local tangent-frame ( $z_{\perp}$ -axis parallel to the gravity vector). Dark blue lines in <i>Figure 1</i>
Body-Frame	B	Frame aligned with the body-frame. $x_{\perp}$ -axis lies in the plane formed by the $x_{\perp}$ and $z_{\perp}$ -axes. Light blues lines in <i>Figure 1</i>

*Figure 1* shows the relative orientation of three of the four frames listed in *Table 1* (ECEF not shown) for a hypothetical body on the surface of the Earth with a roll of 20°, a pitch of 10°, and a heading of 30°. The dashed red lines illustrate the components of the  $\perp$ -frame axes in the N-Frame while the dashed blue lines indicate the projection of the B-Frame axes onto the N-frame.

<sup>1</sup> Strictly speaking, neither the ECEF-frame nor the NED-frame are inertial. Both move and rotate relative to the stars; the NED-frame changes with location as well. However, the two are sufficient for use with the OpenIMU line of products.

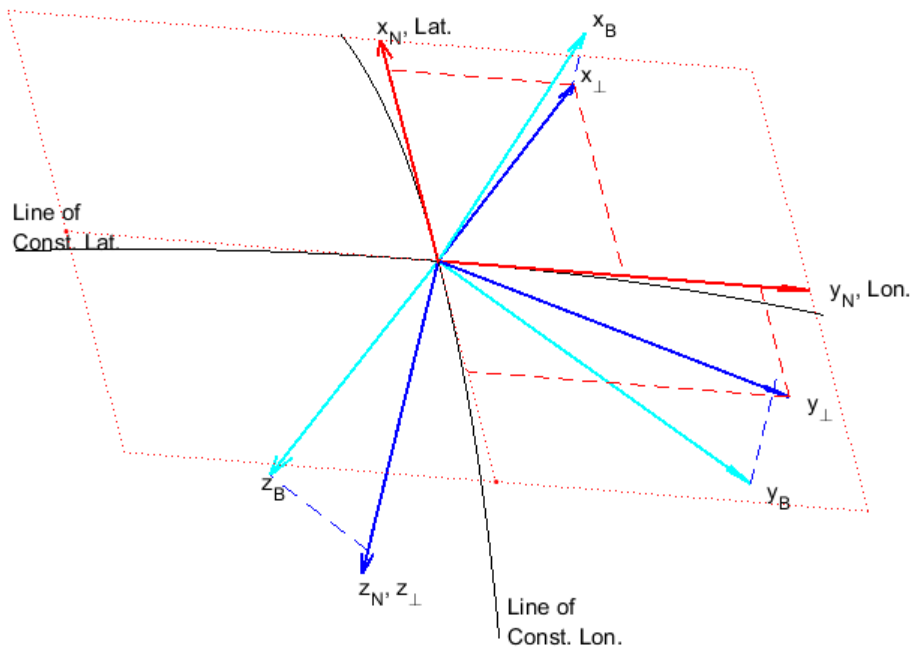


Fig. 1: Figure 1: Coordinate Frames used in Derivation (N, perp, and B-Frames)

## 14.2 Attitude Parameters

### Contents

- *Direction Cosine Matrices*
- *Quaternion Elements*
- *Euler Angles*
- *Mathematical Relationships between Attitude Parameters*
- *Attitude Parameters Example*

This paper makes use of three different attitude parameters to specify the orientation of a body (B) relative to another frame (such as the N-frame).

1. Direction Cosine Matrices
2. Quaternion Elements
3. Euler Angles

### 14.2.1 Direction Cosine Matrices

The first of these, the direction cosine matrix<sup>1</sup>,  ${}^N R^B$ , specifies the relationship of one frame relative to another by relating how the basis-vectors of one frame relate to the basis-vectors of another. These matrices have the property

<sup>1</sup> Pronounced "R B-in-N" and refers to the orientation of the B-Frame in the N-Frame. Also referred to as a rotation transformation matrix.

that they can, in a straightforward manner, transform vectors from one frame into another, such as from the Body to the NED-frame:

$$\vec{x}^N = {}^N R^B \cdot \vec{x}^B$$

In the upcoming derivation, transformations based on the Body-Fixed 3-2-1 Rotation set<sup>2</sup> and the formulation used by Thomas Kane<sup>3</sup> are relied upon extensively.

## 14.2.2 Quaternion Elements

The second parameter used to convey orientation information are quaternion elements<sup>4</sup> (also called Euler parameters),  ${}^N \vec{q}^B$ . Quaternions are relatively easy to propagate in time and do not possess singularities. However, they are not intuitive. Quaternions consist of a scalar and a vector component:

$$\begin{aligned} {}^N \vec{q}^B &= [q_0 \quad \vec{q}_v]^T \\ &= [q_0 \quad q_1 \quad q_2 \quad q_3]^T \\ &= \left[ \cos\left(\frac{\theta}{2}\right) \quad \hat{u} \cdot \sin\left(\frac{\theta}{2}\right) \right]^T \end{aligned}$$

## 14.2.3 Euler Angles

The final parameter used to relay attitude information are Euler angles. These are more intuitive than quaternions but, unlike quaternions, experience singularities at certain angles (based on the selected rotation sequence). For a 321-rotation sequence<sup>5</sup>, the singularity occurs at a pitch of 90°.

## 14.2.4 Mathematical Relationships between Attitude Parameters

All three parameters contain the same information. The equations that relate the various parameters follow<sup>6</sup>. For a 321-rotation sequence, the expression relating the rotation transformation matrix of the body-frame in the NED-frame,  ${}^N R^B$ , to the quaternion elements,  ${}^N \vec{q}^B$ , is:

$${}^N R^B = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2 \cdot (q_1 \cdot q_2 - q_0 \cdot q_3) & 2 \cdot (q_1 \cdot q_3 + q_0 \cdot q_2) \\ 2 \cdot (q_1 \cdot q_2 + q_0 \cdot q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2 \cdot (q_2 \cdot q_3 - q_0 \cdot q_1) \\ 2 \cdot (q_1 \cdot q_3 - q_0 \cdot q_2) & 2 \cdot (q_2 \cdot q_3 + q_0 \cdot q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}$$

<sup>2</sup> A 3-2-1 rotation set defines the attitude of one set of basis-vectors (local-frame) relative to another by specifying the angles of rotation required to get from the inertial (N) to the local-frame (L). With the local and inertial-frames initially aligned, the rotations are performed in the following order: the first is about the local z-axis (3), followed by a rotation about the local y-axis (2), and finally by a rotation about the local x-axis (1). The resulting matrix,  ${}^N R^L = R_{321}$ , is composed of column vectors formed from the xyz-axes of the local-frame coordinatized in the inertial-frame:  ${}^N R^L = [\hat{x}_L^N \quad \hat{y}_L^N \quad \hat{z}_L^N]$ .

<sup>3</sup> Kane, Thomas R.; Levinson, David A. (1985), Dynamics, Theory and Applications, McGraw-Hill series in mechanical engineering, McGraw Hill. Note: one main difference between Kane's approach is the DCM is the transpose of the DCM of other formulations; I think Kane's formulation is more intuitive.

<sup>4</sup> Commonly referred to simply as "quaternion". To make it easier to reference the elements in c, c++, and python, the first quaternion-element (the scalar component of the quaternion) will have the zero index and is expressed as  $q_0 = \cos(\theta/2)$ . The vector component of the quaternion,  $\vec{q}_v = \hat{u} \cdot \sin(\theta/2)$ , occupies elements 2, 3, and 4.

<sup>5</sup> The 321-rotation sequence is the only rotation sequence considered in this paper.

<sup>6</sup> Based on unpublished notes by Keith Reckdahl (Direction Cosines, Rotations, and Quaternions); this paper follows Kane's approach closely. Any reference on the subject will work.

${}^N R^B$  can also be expressed in terms of Euler-angles,  ${}^N \vec{\Theta}^B = [\phi^B \quad \theta^B \quad \psi^B]^T$ :

$${}^N R^B = \begin{bmatrix} \cos(\phi^B) & -\sin(\phi^B) & 0 \\ \sin(\phi^B) & \cos(\phi^B) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta^B) & \sin(\theta^B) \cdot \sin(\psi^B) & \sin(\theta^B) \cdot \cos(\psi^B) \\ 0 & \cos(\psi^B) & -\sin(\psi^B) \\ -\sin(\theta^B) & \cos(\theta^B) \cdot \sin(\psi^B) & \cos(\theta^B) \cdot \cos(\psi^B) \end{bmatrix}$$

In this case,  ${}^N R^B$  is broken up into two sequential transformations, which separate the roll and pitch calculations from the heading (this method is used later to form attitude measurements from the accelerometer and magnetometer readings):

$${}^N R^B = {}^N R^\perp \cdot {}^\perp R^B$$

Finally, Euler angles,  ${}^N \vec{\Theta}^B$ , can be expressed in terms of quaternion-elements,  ${}^N \vec{q}^B$ :

$$\phi^B = \text{atan2}(2 \cdot (q_2 \cdot q_3 + q_0 \cdot q_1), q_0^2 - q_1^2 - q_2^2 + q_3^2)$$

$$\theta^B = -\text{asin}(2 \cdot (q_1 \cdot q_3 - q_0 \cdot q_2))$$

$$\psi^B = \text{atan2}(2 \cdot (q_1 \cdot q_2 + q_0 \cdot q_3), q_0^2 + q_1^2 - q_2^2 - q_3^2)$$

**Note:** Due to the way the roll and pitch are separated from the heading, the Euler angles,  $\phi^B$ ,  $\theta^B$ , and  $\psi^B$  are the same if written as  ${}^N \phi^B$ ,  ${}^N \theta^B$ , and  ${}^N \psi^B$ .

### 14.2.5 Attitude Parameters Example

Using the direction cosine matrix formulation, the transformation to get from the body to inertial-frame (ECEF) in *Figure 1* is composed of multiple transformations:

$${}^E R^B = {}^E R^N \cdot {}^N R^\perp \cdot {}^\perp R^B$$

Each transformation describes how one coordinate frame is related to the next in the sequence of rotations.

1.  ${}^\perp R^B$ : Transformation from the (light-blue) body-frame to the (dark blue) local perpendicular-frame ( $\perp$ )
2.  ${}^N R^\perp$ : Transformation from the (dark blue)  $\perp$ -frame to the (red) local NED-frame
3.  ${}^E R^N$ : Transformation from the (red) NED-frame to the ECEF-frame (ECEF-Frame not shown; black line are latitude and longitude lines).  ${}^E R^N$  is based on the WGS84 model.

This notation not only makes the formulation easier by simplifying the full complexity of the transformation but it helps avoid confusion by explicitly specifying the frame used in each calculation.

Some additional information about these frames:

1.  ${}^E R^N$ , the transformation between the NED and Earth-frame (used in the INS formulation), is solely a function of ECEF location,  ${}^E R^N = f(\vec{r}^E)$ , and is based on the WGS84 model.
2.  ${}^N R^B$ , the transformation between the NED and body-frame is solely a function of the attitude of the body-frame (roll, pitch, and heading angles of the body) and can be measured by the local gravity and magnetic-field vectors (or GPS heading),  ${}^N R^B = f(\vec{g}, \vec{b})$

## 14.3 Sensors

Various sensors are used to obtain the information needed to estimate the position, velocity, and attitude of a system (Table 2). Measurements from these sensors, taken over time, are combined using an Extended Kalman Filter (EKF) to arrive at estimates that are more accurate or more timely than ones based on any single measurement.

Table 2: **Table 2: Inertial Sensors and Measurement Type**

Measurement	Sensor	Description
Position	GPS	GPS provides position (Latitude/Longitude/Altitude) and supplemental information (like standard deviation) to the algorithm. This is used to update the errors in the position (integrated velocity) estimate.
Velocity	1) Accelerometer 2) GPS	Accelerometers provide the high DR/low-noise signal that is integrated to get high DR velocity information. GPS provides velocity and supplemental information to the algorithm (velocity, heading, latency, etc), which is used to update errors due to integration of the accelerometer signal (in particular, to estimate the accelerometer bias).
Roll/Pitch	1) Angular-Rate Sensor 2) Accelerometer	Angular-rate sensors provide the high DR/low-noise signal that is integrated to get high DR attitude information. Accelerometers are used as a gravity reference to update errors due to integration of the rate-sensor signal (in particular, to estimate the rate-sensor bias).
Heading	1) Angular-Rate Sensor 2) Magnetometer 3) GPS	Angular-rate sensors provide the high DR/low-noise signal that is integrated to get high DR heading information. Magnetometers are used as a north-reference to update errors due to integration of the rate-sensor signal (in particular, to estimate the z-axis rate-sensor bias). GPS also provides heading information, which is used in lieu of magnetometer readings and can be more accurate (less prone to disturbances) than the magnetometer but available less often.

Other sensors, such as odometers, barometers, cameras, etc., may be incorporated into the EKF formulation to get improved results. However, incorporating data from any additional sensors would require a reformulation of the algorithm presented here.

Inertial sensors measure the true motion and attitude of a system, corrupted by bias, noise, and external influences. For instance, the accelerometer signal is a combination of platform motion and gravity<sup>1</sup>, as well as sensor bias and noise. Simplified equations for the three sensors are provided below:

$$\vec{\omega}_{meas} = \vec{\omega}_{true} + \vec{\omega}_{bias} + \vec{\omega}_{noise}$$

$$\vec{a}_{meas} = \vec{a}_{motion} + \vec{a}_{grav} + \vec{a}_{bias} + \vec{a}_{noise}$$

$$\vec{m}_{meas} = \vec{b}_{motion} + \vec{m}_{bias} + \vec{m}_{noise}$$

Items, such as misalignment, cross-coupling, etc. are ignored in this formulation they are accounted for during system calibration.

Additionally, sensor bias can be broken down further. In this paper, bias is modeled as a constant offset plus random drift:

$$\vec{\omega}_{bias} = \vec{\omega}_{offset} + \vec{\omega}_{drift}$$

The magnetic field vector,  $\vec{b}$ , may be corrupted by hard and soft-iron sources present in the system in which the part is installed. Hard and soft-iron effects can be estimated by performing a “magnetic-alignment”<sup>2</sup> procedure once installed in the end-user’s system. The equations relating the hard and soft-iron effects<sup>3</sup> on the measured magnetic field is:

$$\vec{m}_{meas} = (R_{SI} \cdot S_{SI} \cdot R_{SI}^T)^{-1} \cdot \vec{b} + \vec{m}_{HI} + \vec{m}_{bias} + \vec{m}_{noise}$$

Where  $R_{SI}$  and  $S_{SI}$  represent the rotation and scaling of the magnetic-field,  $\vec{b}$ , due to soft-iron effects;  $\vec{m}_{HI}$  is the bias change in the magnetic-field due to hard-iron in the system. Sensor gain is measured during the calibration process with the system at room temperature; it does not vary much over temperature. Sensor bias, however, is strongly linked to temperature. The calibration process measures bias over temperature (from -40° C to +85° C). The temperature effect on the magnetometer is “ratiometric”; the unitized magnetic-field vector is unaffected by temperature.

Finally, and most importantly for the Extended Kalman Filter application, all sensor noise signals are assumed to be white, Gaussian, stationary, and independent. This implies that a sensor’s noise characteristics are:

- zero-mean ( $\mu = 0$ )
- distributed according to a normal distribution with variance  $\sigma^2$
- constant over time ( $\sigma^2 \neq f(t)$ )
- uncorrelated with other signals ( $E[(\sigma_{\omega,x} - E[\sigma_{\omega,x}]) \cdot (\sigma_{\omega,y} - E[\sigma_{\omega,y}])] = 0$ )

The formulation of the covariance matrices relies heavily on these assumption.

---

**Note:** The process-noise vectors,  $\vec{w}$ , result from sensor noise transmission through the individual state-transition models, described in the sections to come.

---

<sup>1</sup> Due to the way the accelerometer measures acceleration, gravity appears like a deceleration and, as such,  $\vec{a}_{grav} = -\vec{g}$ . This is gravity deflecting the proof-mass in the direction of the gravity vector; such a deflection caused solely by acceleration would require the body to accelerate in the negative direction.

<sup>2</sup> During a magnetic alignment maneuver, the magnetic measurements are recorded as the system rotates (about its z-axis) through 360 deg. Upon completion of the maneuver, a best-fit ellipse is determined and used to model the hard and soft-iron distortions in the system (described later).

<sup>3</sup> In general you want the magnetic sensor to be in as magnetically clean a location as possible. Even by correcting for hard and soft-iron using this relationship, large hard and soft-iron errors lead to progressively worse solutions.

## 14.4 Kalman Filter

### Contents

- *Prediction (High Dynamic Range (DR) Process)*
- *Innovation (Measurement Error)*
- *Update (Low DR Process)*

The solution described in this document is based on a Kalman Filter that generates estimates of attitude, position, and velocity from noisy sensor readings. The classic Kalman Filter works well for linear models, but not for non-linear models. Therefore, an Extended Kalman Filter (EKF) is used due to the nonlinear nature of the process and measurements model.

Kalman filters operate on a predict/update cycle<sup>1</sup>. The system state at the next time-step is estimated from current states and system inputs. For attitude calculations, this input is the angular rate-sensor signal; velocity and position calculations use the accelerometer signal. The update stage corrects the state estimates for errors inherent in the measurement signals (such as sensor bias and drift) using measurements of the true attitude, position, and velocity estimated from the accelerometer, magnetometer, and GPS readings. As these signals are typically noisier<sup>2</sup> or provided at a significantly lower rate than the rate-sensor, they are not used to propagate the attitude, instead their information is used to correct the errors in the estimate.

For a discrete-time system the prediction and update equations are:

### 14.4.1 Prediction (High Dynamic Range (DR) Process)

In this stage of the EKF, the attitude, velocity, and acceleration are propagated forward in time from sensor readings.

$$\vec{x}_{k|k-1} = f(\vec{x}_{k-1|k-1}, \vec{u}_{k|k-1})$$

$$P_{k|k-1} = F_{k-1} \cdot P_{k-1|k-1} \cdot F_{k-1}^T + Q_{k-1}$$

The first equation ( $\vec{x}_{k|k-1}$ ) is the State Prediction Model and the second ( $P_{k|k-1}$ ) is the Covariance Estimate.

### 14.4.2 Innovation (Measurement Error)

In this stage, the errors between the predicted states and the measurements are computed.

$$\vec{v}_k = \vec{z}_k - \vec{h}_k$$

<sup>1</sup> Kalman Filtering: Theory and Practice Using MATLAB, 3rd Edition, Mohinder S. Grewal, Angus P. Andrews

<sup>2</sup> In this case, noisier means that the sensor signals are corrupted, not just by electrical noise, but by external influences as well. In the case of the accelerometer, the device picks up vehicle motion in addition to gravity information. The magnetometer signal is affected by external magnetic sources, such as iron in passing vehicles and in roadways.



### 14.4.3 Update (Low DR Process)

The final stage of the EKF generates updates (corrections) to the predictions based on the quality of the process models, process inputs, and measurements.

$$\begin{aligned}
 S_k &= H_k \cdot P_{k|k-1} \cdot H_k^T + R_k \\
 K_k &= P_{k|k-1} \cdot H_k^T \cdot S_k^{-1} \\
 \Delta \vec{x}_k &= K_k \cdot \vec{v}_k \\
 \vec{x}_{k|k} &= \vec{x}_{k|k-1} + \Delta \vec{x}_k \\
 \Delta P_k &= -K_k \cdot H_k \cdot P_{k|k-1} \\
 P_{k|k} &= P_{k|k-1} + \Delta P_k
 \end{aligned}$$

In the order listed, the above equations relate to:

1. Innovation Covariance
2. Kalman Gain
3. State Error
4. State Update
5. Covariance Error
6. Covariance Update

These terms will be defined in the sections that follow.

## 14.5 State Transition Models

### 14.5.1 System State-Transition Model Summary<sup>1</sup>

The state-transition models form the core of the EKF prediction stage by performing the following roles:

- 1) They form the equations that propagate the system states from one time-step to the next (using high-quality sensor as the input)
- 2) They define the process-noise vectors relating each state to sensor noise
- 3) They enable computation of the process covariance matrix, Q, and process Jacobian, F. Both are used to propagate the system covariance, P, from one time-step to the next.

The complete system state equation consists of 16 total states<sup>2</sup>

$$\vec{x} = \begin{Bmatrix} \vec{r}^N \\ \vec{v}^N \\ {}^N\vec{q}^B \\ \vec{\omega}_{bias}^B \\ \vec{a}_{bias}^B \end{Bmatrix} = \begin{Bmatrix} \text{NED Position (3)} \\ \text{NED Velocity (3)} \\ \text{Body Attitude (4)} \\ \text{Angular-Rate Bias (3)} \\ \text{Accelerometer Bias (3)} \end{Bmatrix}$$

with the state-transition model,  $\vec{f}$ , made up of five individual models (developed in upcoming sections):

$$\vec{x}_k = \vec{f}(\vec{x}_{k-1}, \vec{u}_{k-1}) + \vec{w}_{k-1}$$

<sup>1</sup> There are many papers describing the derivation and implementation issues for EKFs and Complementary-Filters. Several of the papers similar to this implementation are referenced in the *Reference* section.

<sup>2</sup> GPS measurements are in latitude/longitude/altitude. These are converted to position in the Earth-frame,  $\vec{r}^E$ . Position in the NED-frame is calculated from the initial starting point at system startup. The state estimate is generated by integrating velocity (estimated from accelerometer data).

where  $\vec{x}$  is the state-vector,  $\vec{u}$  is the input-vector (consisting of sensor signals) and  $\vec{w}$  is the process-noise vector.

The expanded state-transition vector,  $\vec{f}$ , is:

$$\vec{f}(\vec{x}_{k-1}, \vec{u}_{k-1}) = \begin{pmatrix} \vec{v}_{k-1}^N + \left[ {}^N R_{k-1}^B \cdot (\vec{a}_{meas,k-1}^B - \hat{a}_{bias,k-1}^B) - \vec{a}_{grav,k-1}^N \right] \cdot dt \\ \left[ I_4 + \frac{dt}{2} \cdot (\Omega_{meas,k-1} - \Omega_{bias,k-1}) \right] \cdot {}^N \vec{q}_{k-1}^B \\ I_3 \\ I_3 \end{pmatrix}$$

and the process-noise vector,  $\vec{w}_{k-1}$ , is:

$$\vec{w}_{k-1} = \begin{pmatrix} -{}^N R_{k-1}^B \cdot \vec{a}_{noise}^B \cdot dt^2 \\ -{}^N R_{k-1}^B \cdot \vec{a}_{noise}^B \cdot dt \\ -\frac{dt}{2} \cdot \Xi_{k-1} \cdot \vec{w}_{noise}^B \\ \vec{N}(0, \sigma_{dd,\omega}^2) \\ \vec{N}(0, \sigma_{dd,a}^2) \end{pmatrix}$$

The sensor noise vectors,  $\vec{N}$ , corresponding to the angular-rate and accelerometer bias states, are each 3x1 vectors with elements described by a zero-mean Gaussian distribution with a variance of either  $\sigma_{dd,\omega}^2$  or  $\sigma_{dd,a}^2$ .

## 14.5.2 Individual State-Transition Models

Individual state-transition models are derived in the following sections:

### Quaternion State-Transition Model

All state propagation equations used in this paper are based on the following Taylor-series expansion:

$$\vec{x}_k = \vec{x}_{k-1} + \dot{\vec{x}}_{k-1} \cdot \frac{dt}{1!} + \ddot{\vec{x}}_{k-1} \cdot \frac{dt^2}{2!} + \dots$$

where terms higher than first-order are neglected. For attitude, the quaternion is propagated according to the expression:

$$\vec{q}_k \approx \vec{q}_{k-1} + \dot{\vec{q}}_{k-1} \cdot dt$$

where  $dt$  is the integration time-step (sampling interval) and  $\vec{q}_{k-1}$  is the current estimate of system attitude.

The kinematical equation that describes the rate-of-change of the attitude quaternion,  $\dot{\vec{q}}_{k-1}$ , is a function of **true angular velocity**,  $\vec{\omega}_{true}$ , as follows:

$$\dot{\vec{q}}_{k-1} = \frac{1}{2} \cdot \Omega_{true,k-1} \cdot \vec{q}_{k-1}$$

where  $\Omega_{true,k-1}$  is formed from the components of the angular rate vector,  $({}^N \vec{\omega}_{true}^B)^B$  and specifies the angular-rate of the body relative to an inertially-fixed frame, measured in the body-frame. As all angular-rate measurements made with MEMS sensors are relative to the inertial-frame, the notation is simplified to  $\vec{\omega}_{true}^B$ .

$$\vec{\omega}^B = \begin{pmatrix} \omega_x^B \\ \omega_y^B \\ \omega_z^B \end{pmatrix}$$

The quaternion propagation matrix,  $\Omega_{k-1}$ , at time-step k-1 is:

$$\Omega_{k-1} = \begin{bmatrix} 0 & -\omega_{x,k-1}^B & -\omega_{y,k-1}^B & -\omega_{z,k-1}^B \\ \omega_{x,k-1}^B & 0 & \omega_{z,k-1}^B & -\omega_{y,k-1}^B \\ \omega_{y,k-1}^B & -\omega_{z,k-1}^B & 0 & \omega_{x,k-1}^B \\ \omega_{z,k-1}^B & \omega_{y,k-1}^B & -\omega_{x,k-1}^B & 0 \end{bmatrix}$$

where (as noted above) all the rate components are estimates of the “true” rate measurements.

From the above expressions, the full state-transition model for system-attitude is:

$$\vec{q}_k = \vec{q}_{k-1} + \frac{1}{2} \cdot \Omega_{true,k-1} \cdot \vec{q}_{k-1} \cdot dt = \left[ I_4 + \frac{dt}{2} \cdot \Omega_{true,k-1} \right] \cdot \vec{q}_{k-1}$$

To find the noise term in the state-transition model,  $\vec{w}_{q,k-1}$ , expand the expression for  $\Omega_{true,k-1}$  using the rate-sensor model described earlier to explicitly show the constituent terms:

$$\Omega_{true,k-1} = \Omega_{meas,k-1} - \Omega_{bias,k-1} - \Omega_{noise,k-1}$$

Substitute this result into the expression for the attitude state-transition model:

$$\begin{aligned} \vec{q}_k &= \left[ I_4 + \frac{dt}{2} \cdot (\Omega_{meas,k-1} - \Omega_{bias,k-1}) - \frac{dt}{2} \cdot \Omega_{noise,k-1} \right] \cdot \vec{q}_{k-1} \\ &= \Phi_{k-1} \cdot \vec{q}_{k-1} + \vec{w}_{q,k-1} \end{aligned}$$

$\Phi_{k-1}$  is the state-transition matrix, defined as:

$$\Phi_{k-1} \equiv I_4 + \frac{dt}{2} \cdot (\Omega_{meas,k-1} - \Omega_{bias,k-1})$$

and  $\vec{w}_{q,k-1}$  is the quaternion process-noise vector:

$$\vec{w}_{q,k-1} = -\frac{dt}{2} \cdot \Omega_{noise,k-1} \cdot \vec{q}_{k-1}$$

---

**Note:** In this expression, the components of  $\Omega_{noise}$  are the noise components of the angular-rate signal,  $\sigma_\omega^2$ . This can be expressed in terms of the sensor’s Angular Random Walk (ARW).

---

Recasting  $\vec{w}_{q,k-1}$ , so the rate-sensor noise ( $\omega_{noise}^B$ ) forms the input vector, results in the final expression for the quaternion process-noise resulting from rate-sensor noise:

$$\vec{w}_{q,k-1} = -\frac{dt}{2} \cdot \Xi_{k-1} \cdot \vec{\omega}_{noise}^B$$

with the variable  $\Xi_{k-1}$  relating the change in process noise to system attitude

$$\Xi_{k-1} \equiv \begin{bmatrix} -\vec{q}_v^T \\ q_0 \cdot I_3 + [\vec{q}_v \times] \end{bmatrix}$$

and  $[\vec{q}_v \times]$  is the cross-product matrix.

The quaternion process noise vector is used to form the elements of the process covariance matrix (Q) related to the attitude state. The covariance is computed according to the following equation:

$$\Sigma_{ij} = cov(\vec{x}_i, \vec{x}_j) = E[(\vec{x}_i - \mu_i) \cdot (\vec{x}_j - \mu_j)]$$

As mentioned previously, all processes considered in this paper assume white (zero mean) sensor noise that is uncorrelated across sensor channels. This simplifies the expression for the covariance to:

$$\Sigma_q = \vec{w}_{q,k-1} \cdot \vec{w}_{q,k-1}^T$$

In addition to the assumption that the noise terms are white and independent, all axes are assumed to have the same noise characteristics ( $\sigma_\omega$ ). Resulting in the final expression for  $\Sigma_q$ :

$$\Sigma_q = \left( \frac{\sigma_\omega \cdot dt}{2} \right)^2 \cdot \begin{bmatrix} 1 - q_0^2 & -q_0 \cdot q_1 & -q_0 \cdot q_2 & -q_0 \cdot q_3 \\ -q_0 \cdot q_1 & 1 - q_1^2 & -q_1 \cdot q_2 & -q_1 \cdot q_3 \\ -q_0 \cdot q_2 & -q_1 \cdot q_2 & 1 - q_2^2 & -q_2 \cdot q_3 \\ -q_0 \cdot q_3 & -q_1 \cdot q_3 & -q_2 \cdot q_3 & 1 - q_3^2 \end{bmatrix}$$

## Velocity State-Transition Model

The velocity propagation equation is based on the following first-order model:

$$\vec{v}_k = \vec{v}_{k-1} + \dot{\vec{v}}_{k-1} \cdot dt$$

$\dot{\vec{v}}_{k-1}$  is an estimate of system acceleration (linear-acceleration corrected for gravity) and is formed from the accelerometer signal with estimated accelerometer-bias and gravity removed.

$$\vec{a}_{motion,k-1} = \vec{a}_{meas,k-1} - \vec{a}_{bias,k-1} - \vec{a}_{grav}$$

Substituting this expression (along with the noise term) into the velocity propagation equation, and explicitly stating the frames in which the readings are made, leads to:

$$\vec{v}_k^N = \vec{v}_{k-1}^N + (\vec{a}_{motion,k-1}^N - {}^N R_{k-1}^B \cdot \vec{a}_{noise}^B) \cdot dt$$

where

$$\vec{a}_{motion,k-1}^N = {}^N R_{k-1}^B \cdot (\vec{a}_{meas,k-1}^B - \hat{a}_{bias,k-1}^B) - \vec{a}_{grav}^N$$

The velocity process-noise vector resulting from accelerometer noise is:

$$\vec{w}_{v,k-1}^N = -{}^N R_{k-1}^B \cdot \vec{a}_{noise}^B \cdot dt$$

leading to the final formulation for the velocity state-transition model:

$$\vec{v}_k^N = \vec{v}_{k-1}^N + \vec{a}_{motion,k-1}^N \cdot dt + \vec{w}_{v,k-1}^N$$

The velocity process noise vector is used to compute the elements of the process covariance matrix ( $Q$ ) related to the velocity estimate, as follows:

$$\Sigma_v = \vec{w}_{v,k-1} \cdot \vec{w}_{v,k-1}^T$$

By making the assumption that all axes have the same noise characteristics ( $\sigma_a^2$ ) and manipulating the expression, the result can be simplified to the following:

$$\Sigma_v = (\sigma_a \cdot dt)^2 \cdot I_3$$

## Position State-Transition Model

The position process model is based on the following first-order model:

$$\vec{r}_k = \vec{r}_{k-1} + \dot{\vec{r}}_{k-1} \cdot dt$$

where  $\dot{\vec{r}}_{k-1}$  is the estimated velocity state,  $\vec{v}_{k-1}$ . Substituting in the velocity term (including noise) results in:

$$\vec{r}_k = \vec{r}_{k-1} + \vec{v}_{k-1} \cdot dt + \vec{w}_{r,k-1}$$

$\vec{w}_{r,k-1}$  is the process noise associated with the position state-transition model, which is directly related to the velocity process noise:

$$\vec{w}_{r,k-1} = \vec{w}_{v,k-1} \cdot dt$$

$$= {}^N R_{k-1}^B \cdot \vec{a}_{noise}^B \cdot dt^2$$

Like the previous process models, this expression is used to compute the elements of the process covariance matrix ( $Q$ ) related to the position estimate:

$$\Sigma_r = \vec{w}_{r,k-1} \cdot \vec{w}_{r,k-1}^T$$

By making the assumption that all axes have the same noise characteristics ( $\sigma_a^2$ ),  $\Sigma_r$  simplifies to:

$$\Sigma_r = (\sigma_a \cdot dt^2)^2 \cdot I_3$$

## Rate and Acceleration Bias State-Transition Models

The process models for the bias terms are based on the assumption that bias is made up of two components:

- 1) A **constant** bias offset ( $\vec{\omega}_{offset}^B$ )
- 2) A **randomly varying** component superimposed on the offset ( $\vec{\omega}_{drift}^B$ ) based on the measured bias-instability value of the sensor

For the rate-sensor, the bias model is

$$\vec{\omega}_{bias}^B = \vec{\omega}_{offset}^B + \vec{\omega}_{drift}^B$$

The drift model follows a random-walk process<sup>1</sup>, i.e. the drift value wanders according to a Gaussian distribution.

$$\vec{\omega}_{drift,k}^B = \vec{\omega}_{drift,k-1}^B + \dot{\vec{\omega}}_{drift,k-1}^B \cdot dt$$

where

$$\dot{\vec{\omega}}_{drift,k-1}^B = N(0, \sigma_{dd,\omega}^2)$$

---

**Note:** The subscript *dd* stands for “drift-dot”.

---

Based on this model, the process variance for  $\vec{\omega}_{drift}^B$  at time, *t*, is given by:

$$\sigma_{d,\omega}^2(t) = [(\sigma_{dd,\omega} \cdot \sqrt{dt}) \cdot \sqrt{t}]^2$$

An empirical study related  $\sigma_{dd,\omega}$  to the BI and ARW values as follows:

$$\sigma_{dd,\omega} = \frac{2 \cdot \pi}{\ln(2)} \cdot \frac{BI^2}{ARW}$$

To find the rate-bias process-noise covariance, set  $t = dt$  in the process-variance model (above), resulting in:

$$\Sigma_{\omega b} = \sigma_{d,\omega}^2(dt) \cdot I_3 = (\sigma_{dd,\omega} \cdot dt)^2 \cdot I_3$$

The accelerometer drift model mirrors this formulation and results in:

$$\Sigma_{ab} = \sigma_{d,a}^2(dt) \cdot I_3 = (\sigma_{dd,a} \cdot dt)^2 \cdot I_3$$

## 14.6 Process Models

### 14.6.1 Introduction

As the state-transition model is nonlinear, the state-transition vector cannot be directly used to propagate the covariance forward in time. Instead the state-transition vector,  $\vec{f}$ , is linearized based on the current system states and used for this task. The resulting linearization (computed from the partial derivatives of  $\vec{f}$  with respect to the system states,  $\vec{x}$ ) generates a matrix referred to as the Process Jacobian,  $F$ . This matrix is used to propagate the covariance,  $P$ , forward in time.

The covariance estimate is also affected by the process noise, which is related to sensor-noise levels. The more process noise that exists in a system, the larger the covariance estimate will be at the next time step. This noise is reflected in the process-noise covariance matrix,  $Q$ .

Formulation of these matrices are described in the following sections.

---

<sup>1</sup> This is not a perfect assumption as the output of the model is unbounded while the actual process is not.

## 14.6.2 Individual Process Models

### Process Jacobian

As the system is nonlinear, the vector  $\vec{f}$  cannot be used to propagate the covariance matrix,  $P$ . Instead the Process Jacobian,  $F$ , (a linearized version of the state-transition vector) is computed at each time step (based on the current system states) to propagate  $P$  forward in time:

$$F_{k-1} = \left. \frac{\partial \vec{f}}{\partial \vec{x}} \right|_{\vec{x}_{k-1}, \vec{u}_{k-1}}$$

This requires taking the derivative of each state-equation with respect to each state. Each row of the Jacobian corresponds to a specific state-equation; each column of the matrix corresponds to a specific system state. Performing this operation results in:

$$F = I_{16} + \begin{bmatrix} 0_3 & I_3 & 0_{3 \times 4} & 0_3 & 0_3 \\ 0_3 & 0_3 & \partial v \partial q & 0_3 & -^N R^B \\ 0_{4 \times 3} & 0_{4 \times 3} & \frac{1}{2} \cdot \Omega & -\frac{1}{2} \cdot \Xi & 0_{4 \times 3} \\ 0_3 & 0_3 & 0_{3 \times 4} & 0_3 & 0_3 \\ 0_3 & 0_3 & 0_{3 \times 4} & 0_3 & 0_3 \end{bmatrix} \cdot dt$$

The one new term in the matrix,  $\partial v \partial q$  is defined as:

$$\partial v \partial q \equiv 2 \cdot \overline{Q}_F \cdot \begin{bmatrix} 0 & (\vec{a}^B)^T \\ \vec{a}^B & -[\vec{a}^B \times] \end{bmatrix}$$

where  $\overline{Q}_F$  is:

$$\overline{Q}_F = \begin{bmatrix} q_1 & q_0 & -q_3 & q_2 \\ q_2 & q_3 & q_0 & -q_1 \\ q_3 & -q_2 & q_1 & q_0 \end{bmatrix}$$

$$= [\vec{q}_v \quad q_0 \cdot I_3 + [\vec{q}_v \times]]$$

and

$$\vec{a}^B = \vec{a}_{meas}^B - \vec{a}_{bias}^B$$

### Process Noise Covariance Matrix

The process covariance acts as a weighting matrix for the system process. It relates the covariance between the  $i^{th}$  and  $j^{th}$  element of each process-noise vector. It is defined as:

$$\Sigma_{ij} = cov(\vec{x}_i, \vec{x}_j) = E[(\vec{x}_i - \mu_i) \cdot (\vec{x}_j - \mu_j)]$$

A Kalman Filter can be viewed the combination of Gaussian distributions to form state estimates.  $Q$  provides a measure of the width of the Gaussian distribution related to each noise state. The wider the distribution, the more uncertainty exists in the process model. This leads to a state-update that affects the state more than if the model had a tighter distribution, which results in an update having less influence on the particular state.

Based on the state process-noise vectors,  $\vec{w}_k$  (found in previous sections), the Process Noise Covariance Matrix is:

$$Q_k = \begin{bmatrix} \Sigma_r & 0_3 & 0_{3 \times 4} & 0_3 & 0_3 \\ 0_3 & \Sigma_v & 0_{3 \times 4} & 0_3 & 0_3 \\ 0_{4 \times 3} & 0_{4 \times 3} & \Sigma_q & 0_{4 \times 3} & 0_{4 \times 3} \\ 0_3 & 0_3 & 0_{3 \times 4} & \Sigma_{\omega b} & 0_3 \\ 0_3 & 0_3 & 0_{3 \times 4} & 0_3 & \Sigma_{ab} \end{bmatrix}$$

The individual process covariance are repeated here:

$$\begin{aligned}\Sigma_r &= (\sigma_a \cdot dt^2)^2 \cdot I_3 \\ \Sigma_v &= (\sigma_a \cdot dt)^2 \cdot I_3 \\ \Sigma_q &= \left(\frac{\sigma_\omega \cdot dt}{2}\right)^2 \cdot \begin{bmatrix} 1 - q_0^2 & -q_0 \cdot q_1 & -q_0 \cdot q_2 & -q_0 \cdot q_3 \\ -q_0 \cdot q_1 & 1 - q_1^2 & -q_1 \cdot q_2 & -q_1 \cdot q_3 \\ -q_0 \cdot q_2 & -q_1 \cdot q_2 & 1 - q_2^2 & -q_2 \cdot q_3 \\ -q_0 \cdot q_3 & -q_1 \cdot q_3 & -q_2 \cdot q_3 & 1 - q_3^2 \end{bmatrix} \\ \Sigma_{\omega b} &= (\sigma_{dd,\omega} \cdot dt)^2 \cdot I_3 \\ \Sigma_{ab} &= (\sigma_{dd,a} \cdot dt)^2 \cdot I_3\end{aligned}$$

## 14.7 Measurement Model

### Overview

It is possible to choose among various measurement models for a given EKF implementation. A particular model is selected based on many factors, one being the limitations of the available measurements. This formulation being described was selected due to the incomplete knowledge of the magnetic environment of the system and uses the available sensor information as follows:

- #. Accelerometers “level” the system (used to compute  ${}^\perp\phi_{meas}^B$  and  ${}^\perp\theta_{meas}^B$ ) FN
- #. Magnetometers and/or GPS heading information align the  $\perp$ -frame with true or magnetic north ( ${}^N\psi^\perp$ )
- #. GPS position and velocity measurements update the position and velocity estimates ( $\vec{r}^N$  and  $\vec{v}^N$ )

Based upon these steps, the measurement vector,  $\vec{z}_k$ , is formed:

$$\vec{z}_k = \begin{Bmatrix} \vec{r}_{GPS}^N \\ \vec{v}_{GPS}^N \\ {}^N\vec{\Theta}_{meas}^B \end{Bmatrix}$$

with the corresponding measurement model,  $\vec{h}_k$ :

$$\vec{h}_k = \begin{Bmatrix} \vec{r}_{pred}^N \\ \vec{v}_{pred}^N \\ {}^N\vec{\Theta}_{pred}^B \end{Bmatrix}$$

Both  ${}^N\vec{\Theta}_{meas}^B$  and  ${}^N\vec{\Theta}_{pred}^B$  are 3x1 column vectors containing the roll, pitch, and heading values.

### Measurement Model

The measurement model,  $\vec{h}_k$  relates the system states,  $\vec{x}_k$ , to the [system measurements](#). The position and velocity elements of this vector come directly from the position and velocity states, while  ${}^N\vec{\Theta}_{pred}^B$  is computed from  ${}^N\vec{q}_{pred}^B$ , as follows:

$${}^\perp\phi_{pred}^B = \text{atan2} \left[ 2 \cdot (q_2 \cdot q_3 + q_0 \cdot q_1), q_0^2 - q_1^2 - q_2^2 + q_3^2 \right]$$

$${}^\perp\theta_{pred}^B = -\text{asin} \left[ 2 \cdot (q_1 \cdot q_3 - q_0 \cdot q_2) \right]$$

$${}^N\psi_{pred}^\perp = \text{atan2} \left[ 2 \cdot (q_1 \cdot q_2 + q_0 \cdot q_3), q_0^2 + q_1^2 - q_2^2 - q_3^2 \right]$$

### Measurement Vector (:math:'\vec{z}\_{\mathbf{k}}')

The measurement vector,  $\vec{z}_k$  is comprised of position, velocity, and attitude information as defined above. It is formed from sensor measurements. However, only the GPS velocity is directly available from measurements; other information must be derived from sensor readings using the relationships described below.

### Roll and Pitch Measurements

Roll and pitch values are computed from the accelerometer signal. Under static conditions, measurements made by the accelerometer consists solely of gravity and sensor noise. Along the axis pointed in the direction of gravity, the sensor measures -1 [g]. This is due to the proof-mass being pulled in the direction of gravity, which, in the absence of gravity, is equivalent to a deceleration of 1 [g].

$$\vec{a}_{meas} = \vec{a}_{grav} = -\vec{g}$$

Static roll and pitch values are determined by noting that gravity is constant in the N-Frame (perp-Frame):

$$\vec{g}^N = \vec{g}^\perp = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}$$

and can be transformed into the body frame through  ${}^B R^\perp$ :

$$\vec{g}^B = {}^B R^\perp \cdot \vec{g}^\perp = ({}^\perp R^B)^T \cdot \vec{g}^\perp = ({}^\perp R^B)^T \cdot \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}$$

Using the definition of  ${}^\perp R^B$  (discussed in [Attitude Parameters](#)) and expanding the equation, the accelerometer measurements can be related to roll and pitch angles:

$$\vec{g}^B = -\vec{a}_{meas}^B$$

$$\begin{Bmatrix} -\sin({}^\perp \theta^B) \\ \cos({}^\perp \theta^B) \cdot \sin({}^\perp \phi^B) \\ \cos({}^\perp \theta^B) \cdot \cos({}^\perp \phi^B) \end{Bmatrix} = \begin{Bmatrix} -a_{mx}^B \\ -a_{my}^B \\ -a_{mz}^B \end{Bmatrix}$$

From this result, the angles corresponding to the accelerometer signal are found:

$${}^\perp \phi_{meas}^B = \text{atan2}(-a_{my}^B, -a_{mz}^B)$$

$${}^\perp \theta_{meas}^B = -\text{asin}(-\hat{a}_{mx}^B)$$

where,  $\hat{a}_{mx}^B$  is the x-axis acceleration value normalized by the total acceleration magnitude:

$$\hat{a}_{mx}^B = \frac{a_{mx}^B}{\|\vec{a}_{meas}^B\|}$$

Normalization of the y and z-axis accelerometer values can be performed. However this is not required as the *atan* function uses the ratio of the two (the normalization factor cancels out).

### Heading Measurements

Heading measurements are determined from one (or both) of the following:

1. Magnetometers
2. GPS Velocity

### Magnetometer-Based Heading



Magnetometers measure the local magnetic field at a high DRs but the readings can be affected by hard and soft-iron disturbances in the system or by changes in the external magnetic field. Hard and soft-iron effects are local to the system and can be accounted for; external field disturbances cannot be corrected.

Adjustment of the magnetic field measurement for hard/soft-iron disturbances can be performed according to the following equation:

$$\vec{m}_{corr}^B = R_{SI} \cdot S_{SI} \cdot R_{SI}^T \cdot (\vec{m}_{meas}^B - \vec{m}_{bias}^B - \vec{m}_{HI}^B)$$

where  $\vec{m}_{meas}^B$  is the measured magnetic field vector in the body-frame,  $\vec{m}_{HI}^B$  is the hard-iron disturbance, and  $R_{SI}$  and  $S_{SI}$  are the soft-iron disturbances.

**Note:** For this analysis the magnetometer bias is neglected; assumed to be negligible or lumped in with the hard-iron.

Hard and soft-iron parameters are estimated by performing a magnetic-alignment maneuver.

**Note:** The application of these corrections do not adjust individual magnetometer channels to match the actual field strength. Only the relative magnetic field is corrected, resulting in a unit-circle for the xy magnetic-field. However, as shown later, this enables the heading to be calculated from the corrected signal.

### Heading calculation

The heading is computed using the fact that, in the magnetic NED-frame, the y-axis component of the magnetic field is zero. In the true-north NED-frame this is not the case; a magnetic declination angle corrects for this. The magnetic field at a given point can be found using the World Magnetic Model (WMM) or from NOAA's website (<https://www.ngdc.noaa.gov/geomag-web/#igrfwmm>). In San Jose, CA, the magnetic field estimates are provided in Table:

Magnetic Field							
Model Used:	WMM2015						
Latitude:	37° 18' 29" N						
Longitude:	121° 50' 51" W						
Elevation:	0.0 km Mean Sea Level						
Date	Declination (+ E   - W)	Inclination (+ D   - U)	Horizontal Intensity	North Comp (+ N   - S)	East Comp (+ E   - W)	Vertical Comp (+ D   - U)	Total Field
2018-07-12	13° 15' 38"	60° 56' 1"	23,328.4 nT	22,706.4 nT	5,351.0 nT	41,970.7 nT	48,018.3 nT
Change/year	-0° 5' 42"/yr	-0° 0' 12"/yr	-44.9 nT/yr	-34.9 nT/yr	-47.9 nT/yr	-86.7 nT/yr	-97.6 nT/yr
Uncertainty	0° 20'	0° 13'	133 nT	138 nT	89 nT	165 nT	152 nT

Fig. 2: Magnetic Field Components based on WMM

Figure illustrates the relationship between the Lines of constant Lat/Lon, the NED-frame, and the perp-frame. Declination is specified with  $\delta$  and heading is specified with  $\psi$ .

The magnetic field vector,  $\vec{b}$ , can be broken down into two components:

- 1) the xy-plane component and
- 2) the vertical component

The relationship between heading and magnetic field is based on the components of  $\vec{b}^N$  as measured in

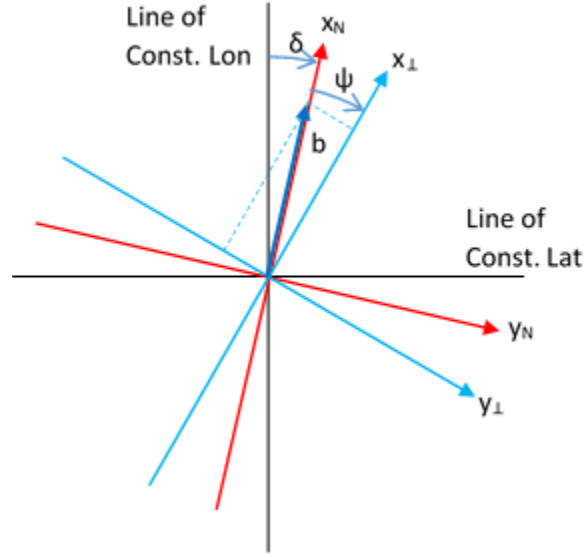


Fig. 3: Relationship of Magnetic-Field to N and B-Frames

the NED-frame:

$$\vec{b}^\perp = {}^\perp R^N \cdot \vec{b}^N = {}^\perp R^N \cdot \begin{Bmatrix} b_{xy} \\ 0 \\ b_z \end{Bmatrix}$$

Expanding the expression results in the following:

$$\begin{Bmatrix} b_x^\perp \\ b_y^\perp \\ b_z^\perp \end{Bmatrix} = \begin{Bmatrix} b_{xy} \cdot \cos({}^N\psi^\perp) \\ -b_{xy} \cdot \sin({}^N\psi^\perp) \\ b_z^\perp \end{Bmatrix}$$

From this, the heading is computed:

$$\tan({}^N\psi^\perp) = \frac{b_{xy} \cdot \sin({}^N\psi^\perp)}{b_{xy} \cdot \cos({}^N\psi^\perp)} = \frac{-b_y^\perp}{b_x^\perp} = \frac{-m_{corr,y}^\perp}{m_{corr,x}^\perp}$$

---

**Note:** The values for  $b_x^\perp$  and  $b_y^\perp$  are the corrected and ‘leveled’ values of the measured magnetic-field in the body-frame; roll and pitch estimates are used to level the signal via  ${}^\perp R_{pred}^B$ .

---

$$\vec{m}_{corr}^\perp = {}^\perp R_{pred}^B \cdot \vec{m}_{corr}^B$$

---

**Note:** As this calculation only corrects the magnetic-field in the xy body-frame, the heading solution is best when the system is nearly level. The solution begins to degrade as the roll and pitch increase. This can be accounted for by adjusting the measurement covariance matrix,  $R$ , accordingly. Additionally, the solution also begins to degrade as the iron in the system increases.

---

## GPS Heading

Heading is also provided directly from the GPS messages. The four messages currently decoded by the IMU381/OpenIMU firmware provide true heading via messages listed in Table.

Table 3: GPS Messaging and Heading Measurement

System	Message	Description	Units
NovAtel	BESTVEL	Actual direction of motion over ground (track over ground) with respect to True North	[deg]
NMEA	VTG	True track made good	[deg]
SiRF	Geodetic Navigation Data – Message ID 41	Course Over Ground (COG, True)	[deg x 100]
ublox	NAV-VELNED	Heading of motion 2-D	[deg]

### 14.7.1 Choosing the Heading Measurement Source

Deciding upon the source of the heading information is ultimately up to the user. In the Aceinna algorithm, the source switches from GPS to magnetometer based on the operating condition. Specifically, during periods of motion, GPS measurements are used as they are considered more accurate as they are not influenced by the magnetic environment. However, when at rest the GPS heading provides no heading information. In this case, the magnetometer provides heading information.

This implementation requires the algorithm to switch not only the source of the data but also the related measurement covariance values.

### GPS Position and Velocity

GPS-based position is derived from the GPS lat/lon/alt message (BestPos, GGA, etc) and converted to NED-position using the WGS84 model.

GPS-based velocity is obtained from the BestVel, etc message. However, the NMEA message does not provide vertical velocity, derived from or accounted for in other ways. In all cases the N and E-velocity is calculated from heading and ground speed. The relationship is:

$$v_N = v_{XY} * \cos({}^N\psi^\perp)$$

$$v_E = v_{XY} * \sin({}^N\psi^\perp)$$

## 14.8 Measurement Vector

### 14.8.1 Model Overview

It is possible to choose among various measurement models for a given EKF implementation. The particular model is selected based on many factors, one being the limitations of the available measurements. This formulation was

selected due to the incomplete knowledge of the magnetic environment of the system and uses the available sensor information as follows:

- 1) Accelerometers “level” the system (used to compute  ${}^\perp\phi_{meas}^B$  and  ${}^\perp\theta_{meas}^B$ ) FN
- 2) Magnetometers and/or GPS heading information align the perp-frame with true or magnetic north ( ${}^N\psi^\perp$ )
- 3) GPS position and velocity measurements update the position and velocity estimates ( $\vec{r}^N$  and  $\vec{v}^N$ )

Based upon these steps, the measurement vector,  $\vec{z}_k$ , is formed:

$$\vec{z}_k = \begin{Bmatrix} \vec{r}_{GPS}^N \\ \vec{v}_{GPS}^N \\ {}^N\vec{\Theta}_{meas}^B \end{Bmatrix}$$

with the corresponding measurement model,  $\vec{h}_k$ :

$$\vec{h}_k = \begin{Bmatrix} \vec{r}_{pred}^N \\ \vec{v}_{pred}^N \\ {}^N\vec{\Theta}_{pred}^B \end{Bmatrix}$$

Both  ${}^N\vec{\Theta}_{meas}^B$  and  ${}^N\vec{\Theta}_{pred}^B$  are 3x1 column vectors containing the roll, pitch, and heading values. FN

## 14.8.2 Measurement Vector ( $\vec{z}_k$ )

The measurement vector,  $\vec{z}_k$  is comprised of position, velocity, and attitude information as defined above. It is formed from sensor measurements. However, only the GPS velocity is available directly from measurements; other information must be derived from sensor readings using the relationship described below.

### Roll and Pitch Measurements

Roll and pitch values are computed from the accelerometer signal. Under static conditions, measurements made by the accelerometer consists solely of gravity and sensor noise. Along the axis pointed in the direction of gravity, the sensor measures -1 [g]. This is due to the proof-mass being pulled in the direction of gravity, which is equivalent to a deceleration of 1 [g] in the absence of gravity.

$$\vec{a}_{meas} = \vec{a}_{grav} = -\vec{g}$$

Static roll and pitch values are determined by noting that gravity is constant in the N-Frame (perp-Frame):

$$\vec{g}^N = \vec{g}^\perp = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}$$

and can be transformed into the body frame through  ${}^B R^\perp$ :

$$\vec{g}^B = {}^B R^\perp \cdot \vec{g}^\perp = ({}^\perp R^B)^T \cdot \vec{g}^\perp = ({}^\perp R^B)^T \cdot \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}$$

Using the definition of  ${}^\perp R^B$  (discussed in [Attitude Parameters](#)) and expanding the equation, the accelerometer measurements can be related to roll and pitch angles:

$$\vec{g}^B = -\vec{a}_{meas}^B$$

$$\begin{Bmatrix} -\sin(\perp\theta^B) \\ \cos(\perp\theta^B) \cdot \sin(\perp\phi^B) \\ \cos(\perp\theta^B) \cdot \cos(\perp\phi^B) \end{Bmatrix} = \begin{Bmatrix} -a_{mx}^B \\ -a_{my}^B \\ -a_{mz}^B \end{Bmatrix}$$

From this result, the angles corresponding to the accelerometer signal are found:

$$\perp\phi_{meas}^B = \text{atan2}(-a_{my}^B, -a_{mz}^B)$$

$$\perp\theta_{meas}^B = -\text{asin}(-\hat{a}_{mx}^B)$$

where,  $\hat{a}_{mx}^B$  is the x-axis acceleration value normalized by the total acceleration magnitude:

$$\hat{a}_{mx}^B = \frac{a_{mx}^B}{\|\vec{a}_{meas}^B\|}$$

Normalization of the y and z-axis accelerometer values can be performed. However this is not required as the *atan* function uses the ratio of the two (the normalization factor cancels out).

## Heading Measurements

Heading measurements are determined from the following:

- 1) Magnetometers
- 2) GPS Velocity

## Magnetometer-Based Heading

Magnetometers measure the local magnetic field at a high DRs but the readings can be affected by hard and soft-iron disturbances in the system or by changes in the external magnetic field. Hard and soft-iron effects are local to the system and can be accounted for; external field disturbances cannot be corrected.

Adjustment of the magnetic field measurement for hard/soft-iron disturbances can be performed according to the following equation:

$$\vec{m}_{corr}^B = R_{SI} \cdot S_{SI} \cdot R_{SI}^T \cdot (\vec{m}_{meas}^B - \vec{m}_{bias}^B - \vec{m}_{HI}^B)$$

where  $\vec{m}_{meas}^B$  is the measured magnetic field vector in the body-frame,  $\vec{m}_{HI}^B$  is the hard-iron disturbance, and  $R_{SI}$  and  $S_{SI}$  are the soft-iron disturbances. Note: for this analysis the magnetometer bias is neglected; assumed to be negligible or lumped in with the hard-iron.

Hard and soft-iron parameters are estimated by performing a magnetic-alignment maneuver. Note that the application of these corrections do not adjust individual magnetometer channels to match the actual field strength. Only the relative magnetic field is corrected, resulting in a unit-circle for the xy magnetic-field. However, as shown later, this enables the heading to be calculated from the corrected signal.

## Heading calculation

The heading is computed using the fact that, in the magnetic NED-frame, the y-axis component of the magnetic field is zero. In the true-north NED-frame this is not the case; a magnetic declination angle corrects for this. The magnetic field at a given point can be found using the World Magnetic Model (WMM) or from NOAA's website (<https://www.ngdc.noaa.gov/geomag-web/#igrfwmm>). In San Jose, CA, the magnetic field estimates are provided in Table 4:

Figure 4 illustrates the relationship between the Lines of constant Lat/Lon, the NED-frame, and the perp-frame. Declination is specified with  $\delta$  and heading is specified with  $\psi$ .

The magnetic field vector,  $\vec{b}$ , can be broken down into two components:

Magnetic Field							
Model Used:	WMM2015						
Latitude:	37° 18' 29" N						
Longitude:	121° 50' 51" W						
Elevation:	0.0 km Mean Sea Level						
Date	Declination ( + E   - W )	Inclination ( + D   - U )	Horizontal Intensity	North Comp ( + N   - S )	East Comp ( + E   - W )	Vertical Comp ( + D   - U )	Total Field
2018-07-12	13° 15' 38"	60° 56' 1"	23,328.4 nT	22,706.4 nT	5,351.0 nT	41,970.7 nT	48,018.3 nT
Change/year	-0° 5' 42"/yr	-0° 0' 12"/yr	-44.9 nT/yr	-34.9 nT/yr	-47.9 nT/yr	-86.7 nT/yr	-97.6 nT/yr
Uncertainty	0° 20'	0° 13'	133 nT	138 nT	89 nT	165 nT	152 nT

Fig. 4: Table 4: Magnetic Field Components based on WMM

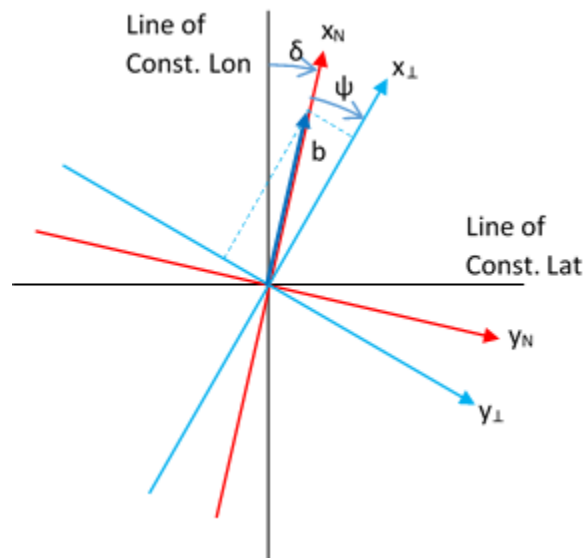


Fig. 5: Figure 4: Relationship of Magnetic-Field to N and B-Frames

- 1) the xy-plane component and
- 2) the vertical component

The relationship between heading and magnetic field is based on the components of  $\vec{b}^N$  as measured in the NED-frame:

$$\vec{b}^\perp = {}^\perp R^N \cdot \vec{b}^N = {}^\perp R^N \cdot \begin{pmatrix} b_{xy} \\ 0 \\ b_z \end{pmatrix}$$

Expanding the expression results in the following:

$$\begin{Bmatrix} b_x^\perp \\ b_y^\perp \\ b_z^\perp \end{Bmatrix} = \begin{Bmatrix} b_{xy} \cdot \cos({}^N\psi^\perp) \\ -b_{xy} \cdot \sin({}^N\psi^\perp) \\ b_z^\perp \end{Bmatrix}$$

From this, the heading is computed:

$$\tan({}^N\psi^\perp) = \frac{b_{xy} \cdot \sin({}^N\psi^\perp)}{b_{xy} \cdot \cos({}^N\psi^\perp)} = \frac{-b_y^\perp}{b_x^\perp} = \frac{-m_{corr,y}^\perp}{m_{corr,x}^\perp}$$

Note: the values for  $b_x^\perp$  and  $b_y^\perp$  are the corrected and ‘leveled’ values of the measured magnetic-field in the body-frame; roll and pitch estimates are used to level the signal via  ${}^\perp R_{pred}^B$ .

$$\vec{m}_{corr}^\perp = {}^\perp R_{pred}^B \cdot \vec{m}_{corr}^B$$

Note: as this calculation only corrects the magnetic-field in the xy body-frame, the heading solution is best when the system is nearly level. The solution begins to degrade as the roll and pitch increase. This can be accounted for by adjusting the measurement covariance matrix,  $R$ , accordingly. Additionally, the solution also begins to degrade as the iron in the system increases.

## GPS Heading

Heading is also provided directly from the GPS messages. The four messages currently decoded by the IMU381/OpenIMU firmware provide true heading via messages listed in Table 6.

Table 4: **Table 6: GPS Messaging and Heading Measurement**

System	Message	Description	Units
NovAtel	BESTVEL	Actual direction of motion over ground (track over ground) with respect to True North	[deg]
NMEA	VTG	True track made good	[deg]
SiRF	Geodetic Navigation Data – Message ID 41	Course Over Ground (COG, True)	[deg x 100]
ublox	NAV-VELNED	Heading of motion 2-D	[deg]

of the PS readings and angles derived from accelerometer readings (equations provided in Measurement Covariance section):

## GPS Position and Velocity

GPS-based position is derived from the GPS lat/lon/alt message (BestPos, GGA, etc) and converted to NED-position using the WGS84 model.

GPS-based velocity is obtained from the BestVel, etc message. However, the NMEA message does not provide vertical velocity, derived from or accounted for in other ways. In all cases the N and E-velocity is calculated from heading and ground speed. The relationship is:

$$\begin{aligned} v_N &= v_{XY} * \cos({}^N\psi^\perp) \\ v_E &= v_{XY} * \sin({}^N\psi^\perp) \end{aligned}$$

## 14.9 Innovation / Measurement Error

### 14.9.1 Innovation Overview

The innovation (measurement error) is formed from the sensor measurements and the predicted states. As the measurements and the system states are often not the same, one or the other needs to be transformed into the measurement. In the case of this algorithm, the state consists of an attitude quaternion, NED-velocity, and NED-position. The measurements come from accelerometer readings, GPS latitude/longitude/altitude measurements, and horizontal/vertical velocities along with ground-track. In this case either the states need to be converted to match the measurements or vice-versa.

Once the measurements vectors are formed, the innovation (measurement error),  $\vec{\nu}_k$ , is computed:

$$\vec{\nu}_k = \vec{z}_k - \vec{h}_k$$

This result is used in the update stage of the EKF to generate the state error,  $\Delta\vec{x}_k$ , given the Kalman gain matrix.

The available sensor information is used as follows:

1. Accelerometers “level” the system (used to compute  ${}^\perp\phi_{meas}^B$  and  ${}^\perp\theta_{meas}^B$ ) FN
2. Magnetometers and/or GPS heading information align the perp-frame with true or magnetic north ( ${}^N\psi^\perp$ )
3. GPS position and velocity measurements update the position and velocity estimates ( $\vec{r}^N$  and  $\vec{v}^N$ )

Measurement Details To Be Provided

## 14.10 Magnetic-Alignment

### Overview

A so-called “magnetic-alignment” procedure enables estimation of the hard and soft-iron disturbances in the system. As these disturbances are fixed in the body, the corrections must be applied in the body-frame. The procedure works as follows:

- 1) The magnetic-field is measured and recorded as the system undergoes a 360+ degree rotation about the z-axis. Ideally this is done when the system is level.
- 2) Upon completion, an algorithm determines the ellipse that best fits the distorted circle.
- 3) Ellipse parameters (related to the hard and soft-iron disturbances) are saved in the firmware and used to correct the magnetic-field measurements.

In most cases an ellipse describes magnetic-field distortions quite well. The ellipse parameters relate to the magnetic disturbances as follows:



- The center of the ellipse is equal to the hard-iron values
- The angle the major-axis of the ellipse makes with a nominal x-axis is equal to the soft-iron angle (which forms the matrix  $R_{SI}$ )
- The major and minor-axis lengths forms the scaling matrix  $S_{SI}$

The formula for the corrected magnetic measurements works by:

- 1) Centering the ellipse by removing the hard-iron bias from the measurements
- 2) Rotating the ellipse to align with the nominal x and y-axes
- 3) Stretching the ellipse along its major and minor-axes to form a unit-circle
- 4) Rotating the unit-circle back into its nominal orientation

Note: as mentioned earlier, this correction is only done in the XY-plane and cannot correct the raw magnetometer signal. It is only done to determine the system heading.

### Example

Magnetic-field information was collected as the system underwent a 360 degree rotation about the z-axis (*Figure*). This was performed twice, once in a disturbance-free environment (no iron added to the system) and once with additional iron added to the system. The data in each case was processed and a best-fit ellipse FN computed (dashed lines). In the disturbance-free case, the data and the fit were close to circular. In the case with additional iron, however, the circle was clearly distorted and shifted away from the origin.

### Magnetic-Field Measurement in an Environment with and without Iron-Based Disturbances

For the measurements taken in the presence of additional iron, the estimation procedure produced the following best-fit ellipse parameters:

#### Best-Fit Ellipse Parameters

Ellipse Parameter	Value	Unit
<i>Center</i>	-0.128, 0.126	[G]
<i>Major/Minor axes</i>	0.225, 0.198	[G]
<i>Soft-Iron Scale Factor</i>	0.882	[N/A]
<i>Angle to Major-Axis</i>	-48.497	[deg]

In the correction equation (above),  $R_{SI}$  is the rotation matrix and corrects for a rotation of the magnetic-field due to soft-iron effects:

$$R_{SI} = \begin{bmatrix} \cos(\eta) & \sin(\eta) & 0 \\ -\sin(\eta) & \cos(\eta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Where  $\eta$  is the angle from the nominal x-axis to the semi-major axis.  $S_{SI}$  (the scale-factor matrix) corrects for the stretching caused by the soft-iron:

$$S_{SI} = \begin{bmatrix} 1/a & 0 & 0 \\ 0 & 1/b & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$a$  and  $b$  are the lengths of the semi-major and semi-minor axes.

For the data-set described above, the values for  $R_{SI}$  and  $S_{SI}$ , resulting from the best-fit ellipse parameters, are:

$$R_{SI} = \begin{bmatrix} 0.66266 & -0.748920 & 0 \\ 0.748920 & 0.662660 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and

$$S_{SI} = \begin{bmatrix} 4.4522600 & 0 & 0 \\ 0 & 5.046890 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Applying these correction factors to the raw magnetic-field measurements results in the unit-circle shown in *Figure*.

### Corrected Magnetic Field Readings

Note: the nodes located at 45 degree increments around the circle are points where additional data was collected to test the heading calculation (described in the next section).

### Results

*Table* lists the heading computed from test data using the above equations relating heading to corrected magnetic-field.

#### Heading Results from Magnetically Clean and Distorted Readings

True Heading [deg]	Disturbance-Free Data		Data with Added Iron Source	
	Heading [deg]	Error [deg]	Heading [deg]	Error [deg]
0	359.69	-0.31	0.013	0.013
45	45.19	0.19	44.82	-0.18
90	89.96	-0.04	90.15	0.15
135	135.05	0.05	135.08	0.08
180	180.57	0.57	180.68	0.68
225	225.64	0.64	225.62	0.62
270	270.63	0.63	270.48	0.48
315	315.30	0.30	315.09	0.09
360	359.79	-0.21	0.10	0.10

Note: the raw results reported a systematic error of approximately 2.0 degrees on all heading values. This was due to a misalignment of the test-fixtue relative to true-north. The values presented in *Table* reflect this 2.0 degree correction. The systematic error is visible in Figures with data-clusters that do not fall on the x and y-axes.

## 14.11 Algorithms References

### Contents

- *General Kalman Filter References*
- *Extended Kalman Filter Implementations*
- *Mathematical References*

### 14.11.1 General Kalman Filter References

Corey Montella. Lehigh University. May 2011. "The Kalman Filter and Related Algorithms: A Literature Review" ([https://www.researchgate.net/publication/236897001\\_The\\_Kalman\\_Filter\\_and\\_Related\\_Algorithms\\_A\\_Literature\\_Review](https://www.researchgate.net/publication/236897001_The_Kalman_Filter_and_Related_Algorithms_A_Literature_Review))

Kalman, R. E. (1960). "A New Approach to Linear Filtering and Prediction Problems". Transaction of the ASME - Journal of Basic Engineering, 35-45. (<https://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf>)

Juler, S., & Uhlmann, J. (n.d.). "A New Extension of the Kalman Filter to Nonlinear Systems" (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.2891&rep=rep1&type=pdf>)

Gordon, N., Salmond, D., & Smith, A. (1993). "Novel approach to nonlinear/non-Gaussian Bayesian state estimation". IEEE Proceedings-F, (pp. 107-113). (<http://www.irisa.fr/aspi/legland/ref/gordon93a.pdf>)

Thrun, S., Burgard, W., & Fox, D. (2005). "Probabilistic Robotics". Cambridge, MA: MIT Press. (<https://docs.ufpr.br/~danielsantos/ProbabilisticRobotics.pdf>)

### 14.11.2 Extended Kalman Filter Implementations

Sabatini, A.M. Quaternion-based extended Kalman filter for determining orientation by inertial and magnetic sensing. IEEE Trans. Biomed. Eng. 2006

### 14.11.3 Mathematical References

#### Process Models

JA Farrell, M. Barth. "The global positioning system & inertial navigation". McGraw-Hill 1998.

A Huster. "Relative position sensing by fusion monocular vision and inertial rate sensors". 2003.

S Julier, J. K. Uhlmann. "A new extension of the Kalman filter to nonlinear system". Proceedings of AeroSense: The 11th International Symposium on Aerospace/Defense Sensing Simulation and Controls Multi Sensor Fusion Tracking and Resource Management II SPIE. 1997.

E Nebot, H. Durrant-Whyte. "Initial calibration and alignment of low-cost inertial navigation units for land vehicle applications". Journal of Robotic Systems, vol. 16, no. 2. 1999.

M Park. "Error analysis and stochastic modeling of MEMS based inertial sensors for land vehicle navigation applications". vol 4. 2004.

OS Salychev, V. V. Voronov, M. E. Cannon, G. Lachapelle. "Low cost INS/GPS integration: concepts and testing". Proceeding of the Institute of Navigation National Technical Meeting. 2000.

S Sukkarieh. "Low cost high integrity aided inertial navigation systems for autonomous land vehicles". 2000.

R van der Merwe, EA Wan. "Sigma-point Kalman filters for integrated navigation". Proceedings of the 60th Annual Meeting of the Institute of Navigation (ION). June 2004.

EA Wan, R. van der Merwe. "The unscented Kalman filter for nonlinear estimation". Symposium 2000 on Adaptive Systems for Signal Processing Communication and Control. vol 10. 2000.

G Welch, G. Bishop. "An introduction to the Kalman filter SIGGRAPH 2001 course 8". Computer Graphics Annual Conference on Computer Graphics & Interactive Techniques. Aug 2001.

#### Measurement Models

Jrg F Wagnera, Thomas Wieneke. "Integrating satellite and inertial navigation - conventional and new fusion approaches". Control Engineering Practice, Volume 11, Issue 5, May 2003. (<https://www.sciencedirect.com/science/article/abs/pii/S0967066102000436>)

Agus Budiyo. "Principles of GNSS, Inertial, and Multi-sensor Integrated Navigation Systems". (2012) "Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems", Industrial Robot: An International Journal, Vol. 39 Iss: 3 ([https://www.researchgate.net/profile/Agus\\_Budiyo/publication/234119885\\_Principles\\_of\\_GNSS\\_Inertial\\_and\\_Multi-sensor\\_Integrated\\_Navigation\\_Systems/links/02e7e51df4ead46a61000000/Principles-of-GNSS-Inertial-and-Multi-sensor-Integrated-Navigation-Systems.pdf](https://www.researchgate.net/profile/Agus_Budiyo/publication/234119885_Principles_of_GNSS_Inertial_and_Multi-sensor_Integrated_Navigation_Systems/links/02e7e51df4ead46a61000000/Principles-of-GNSS-Inertial-and-Multi-sensor-Integrated-Navigation-Systems.pdf))

#### Innovation (Measurement Error)

Drora Goshen-Meskin; Itzhack Y. Bar-Itzhack. "Unified approach to inertial navigation system error modeling". Journal of Guidance, Control, and Dynamics, May 1992, Vol. 15, No. 3 (<https://arc.aiaa.org/doi/pdf/10.2514/3.20887>)

George Arshal. "Error equations of inertial navigation". July 1987. Journal of Guidance, Control, and Dynamics, July 1987, Vol. 10, No. 4 (<https://doi.org/10.2514/3.20225>)

Drora Goshen-Meskin And Itzhack Y. Bar-Itzhack. "Unified approach to inertial navigation system error modeling" journal of Guidance, Control, and Dynamics, May 1992, Vol. 15, No. 3 <https://doi.org/10.2514/3.20887>

#### Magnetic-Alignment

Sabatini, A.M. Quaternion-based extended Kalman filter for determining orientation by inertial and magnetic sensing. IEEE Trans. Biomed. Eng. 2006

Sebastian O.H. Madgwick. "An efficient orientation filter for inertial and inertial-magnetic sensor arrays". April 30, 2010. University of Bristol UK (<https://forums.parallax.com/uploads/attachments/41167/106661.pdf>)

Marins, João LuÃs; Yun, Xiaoping; Bachmann, Eric R.; McGhee, Robert B.; Zyda, Michael J. "An Extended Kalman Filter for Quaternion-Based Orientation Estimation Using MARG Sensors" Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, Maui, Hawaii, USA, Oct. 29 - Nov. 03, 2001 (<https://calhoun.nps.edu/bitstream/handle/10945/41567/IROS2001.pdf;sequence=3>)

#### Other References

Other References - Keith Mitiguy Kane - To Be Provided

---

### Magnetic Sensor Algorithms

---

OpenIMU ships with a number of ready to use, [downloadable applications](#) to help you get started.

This section discusses algorithms that can make use of the OpenIMU's on-board magnetic sensor. Currently, this is primarily for Magnetic Alignment also referred to as Compass Calibration, or Hard/Soft Iron Calibration.

In the future, this section may include other algorithms that make use of the magnetometer including event detection and pedestrian dead reckoning.

**Part IV**

**Miscellaneous**

## CHAPTER 16

---

### C-Code Serial Driver

---

C-code Serial Driver - Details To Be Provided

## E

### environment variable

- ABSOLUTEMAXIMUMRATINGS, 158
- COMPLIANCE, 158
- ELECTRICAL, 126, 158
- ENVIRONMENT, 126, 158
- INPUTVOLTAGETOLERANCE, 159
- PHYSICAL, 126, 158
- VALUES, 158
- VOLTAGEVALUES, 158