Table of contents

<table>
<thead>
<tr>
<th>I</th>
<th>About OpenIMU</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Development Tools</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>II</td>
<td>Quick Start</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>PC Tools Installation</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>OpenIMU300ZI Evaluation Kit Setup</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>OpenIMU300RI Evaluation Kit Setup</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>OpenIMU330BI Evaluation Kit Setup</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>III</td>
<td>IMU Modules</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>OpenIMU Modules</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IV</td>
<td>Evaluation Kits</td>
<td>61</td>
</tr>
<tr>
<td>8</td>
<td>OpenIMU300ZI Eval Kit</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>OpenIMU300RI Eval Kit</td>
<td>71</td>
</tr>
<tr>
<td>10</td>
<td>OpenIMU330BI Eval Kit</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>Software Development</td>
<td>84</td>
</tr>
<tr>
<td>11</td>
<td>Tutorial - What The User Needs to Know to Build The First Application</td>
<td>85</td>
</tr>
<tr>
<td>12</td>
<td>OpenIMU Software Support</td>
<td>120</td>
</tr>
<tr>
<td>13</td>
<td>Ready-to-Use Applications</td>
<td>151</td>
</tr>
</tbody>
</table>
14 EKF Algorithms 155
15 Magnetic Sensor Algorithms 183

VI Miscellaneous 184
16 Algorithm Simulation System 185
17 Python Serial Driver 186
18 C-Code Serial Driver 187

Index 188
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

About OpenIMU
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:

<table>
<thead>
<tr>
<th>Hardware Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td><em>EZ</em></td>
</tr>
<tr>
<td><em>EZ</em></td>
</tr>
<tr>
<td><em>CAN</em></td>
</tr>
<tr>
<td><em>CAN</em></td>
</tr>
<tr>
<td><em>LC</em></td>
</tr>
</tbody>
</table>

Open-Source Embedded Software

• OpenIMU hardware runs an open-source stack written on top of FreeRTOS and standard ARM Cortex libraries.
• 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).
- Additional site features are being actively developed and will include additional analysis tools such as simulation, as well as a user Forum.

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 as a “CLI” command line interface to your customized OpenIMU 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 websocket server connecting the OpenIMU hardware with our ANS developer platform for a GUI experience, cloud data storage and retrieval, and online 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
```

**Device configuration**

- Device options (settings) are exposed on the device configuration page.
- 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 record page to plot live data. Use the log switch to save this data *(Requires Login)*. Data is saved *both* locally and then backed up to the cloud for later retrieval.
File Retrieval

Logged files are retrieved on the My Files page which opens up a zoomable graph view. *Requires Login*
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.
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
- OpenIMU300ZI Evaluation Kit
- OpenIMU300RI Evaluation Kit
• OpenIMU330BI Evaluation Kit
• In System Firmware Update
• Python Interface
• ‘openimu.json’ Configuration File

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
>>server_start

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

Once server_start command has been entered, open the following link https://developers.aceinna.com/connect

The parameters that show up in this table are controlled by openimu.json and their corresponding code in userConfiguration.c. Select the packet that you would like to display.

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

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

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

**Note:**
- The Aceinna VSCode extension uses the underlying PlatformIO debugging feature.
- The user must have at least the initial license for using the PlatformIO platform, which is the free “PlatformIO Plus Community” license. That license provides a 30 day trial of the “PlatformIO Plus Professional” features.
- The debug ability depends on have the “PlatformIO Plus Professional” features.
- **You must have a “PlatformIO Plus Professional” license to continue using the debugger after the first 30 days** ([https://platformio.org/pricing](https://platformio.org/pricing))
- Using more advanced debugging features require a PlatformIO Enterprise license (also at: [https://platformio.org/pricing](https://platformio.org/pricing)).

**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.

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.
2. 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.

1. 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:

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

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

```c
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

2.3. Debugging using the PlatformIO Debugger and the JTAG Debug Adapter
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.

2.4 OpenIMU300ZI Eval Board

2.4.1 OpenIMU300ZI Eval Board and Fixture

The OpenIMU300RA Eval Board is attached to a fixture for easy handling and isolation of the back side of the board from any contact. The OpenIMU300EVB interfaces to the main connector of the OpenIMU300EZ. 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 the OpenIMU300’s serial ports from 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.
2.4.2 OpenIMU300ZI 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**.

![Coordinate System Diagram]

2.5 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 configurable elements are known as **Configuration Parameters**.
Fig. 1: OpenIMU300RI Eval Kit Fixture and Board
2.6 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**

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x4A49</td>
<td>0x00</td>
<td></td>
<td>CRC(U2)</td>
</tr>
</tbody>
</table>

The command allows system to enter bootloader mode.

**Write App Command**

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x5741</td>
<td>len+5</td>
<td></td>
<td>CRC(U2)</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Scaling</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>startingAddr</td>
<td>U4</td>
<td></td>
<td>bytes</td>
<td>The FLASH word offset to begin writing data</td>
</tr>
<tr>
<td>4</td>
<td>byteLength</td>
<td>U1</td>
<td></td>
<td>bytes</td>
<td>The word length of the data to write</td>
</tr>
<tr>
<td>5</td>
<td>dataByte0</td>
<td>U1</td>
<td></td>
<td></td>
<td>Flash data</td>
</tr>
<tr>
<td>6</td>
<td>dataByte1</td>
<td>U1</td>
<td></td>
<td></td>
<td>Flash data</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4+byteLength</td>
<td>dataByte</td>
<td>U1</td>
<td></td>
<td></td>
<td>Flash data</td>
</tr>
</tbody>
</table>

Payload starts from 4-byte address of flash memory where the binary is located. The fifth byte is the number of bytes of dataBytes, 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**
The command allows system directly to enter application mode.

2.7 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.

```python
# 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

(continues on next page)
>>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.

2.8 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 they are sent in userMessaging.c.

```json
{
    "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
→" }
    ]
}
```

(continues on next page)
OpenIMU Documentation

(continued from previous page)

```json
{
  "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"
      }
    },
    {
      "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": "Set Configuration Parameters"
    }
  ]
}
```

(continues on next page)
"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",
"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 }
{ "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": "" } ]}
"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
}
]
},
{
"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"
}]}
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 stared quickly is to purchase an OpenIMU Developer’s Kit from Aceinna [https://www.aceinna.com](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.
Part II

Quick Start
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:

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 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
```
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 “Development Tools” section:
  - Download App with JTAG
  - Debugging with PlatformIO Debugger and JTAG Debug Adapter
  - Graphing & Logging IMU Data using the Acienna Navigation Studio
Fig. 1: OpenIMU300ZI Evaluation Kit
CHAPTER 5

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 “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:
Fig. 1: OpenIMU300RI Evaluation Kit
• 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

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Unit Connector</th>
<th>RS232 Connector</th>
<th>CAN Connector</th>
<th>Power Wires</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>3</td>
<td>5</td>
<td></td>
<td>Black</td>
</tr>
<tr>
<td>VIN</td>
<td>6</td>
<td></td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>RS232 TX</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS232 RX</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN H</td>
<td>1</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>CAN L</td>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**OpenIMU300RI Connector**
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
Fig. 1: OpenIMU330BI Evaluation Kit
Part III

IMU Modules
OpenIMU Modules

There are currently available are the:

### 7.1 OpenIMU300ZI - EZ Embed Automotive Module

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
- Update Rate to 800Hz
- 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

### 7.1.1 IMU Sensors

#### ANGULAR RATE

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: X,Y,Z (°/Sec)</td>
<td>&gt; 400</td>
</tr>
<tr>
<td>Bias Instability (%/Hr)</td>
<td>&lt; 6</td>
</tr>
<tr>
<td>Bias Stability Over Temperature (%/Sec)</td>
<td>&lt; 0.5</td>
</tr>
<tr>
<td>Resolution (°/Sec)</td>
<td>&lt; 0.02</td>
</tr>
<tr>
<td>Scale Factor Accuracy (%)</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>Non-Linearity (%)</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>Angle Random Walk (°/hr)</td>
<td>&lt; 0.3</td>
</tr>
<tr>
<td>Bandwidth (Hz)</td>
<td>5-50Hz (Configurable)</td>
</tr>
</tbody>
</table>

Notes: 1 Allen Variance Curve, Constant Temperature, 2 1-Sigma Error, 3 RMS Error over Temperature

#### ACCELERATION

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: X,Y,Z (g)</td>
<td>&gt; 8</td>
</tr>
<tr>
<td>Bias Instability (g)</td>
<td>&lt; 20</td>
</tr>
<tr>
<td>Bias Stability Over Temperature (mg)</td>
<td>&lt; 5</td>
</tr>
<tr>
<td>Resolution (mg)</td>
<td>&lt; 0.5</td>
</tr>
<tr>
<td>Scale Factor Accuracy (%)</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>Non-Linearity (%)</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>Velocity Random Walk (m/s/hr)</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>Bandwidth (Hz)</td>
<td>5-50Hz (Configurable)</td>
</tr>
</tbody>
</table>
Notes: 1 Allen Variance Curve, Constant Temperature, 2 1-Sigma Error, 3 RMS Error over Temperature

**MAGNETIC FIELD**

<table>
<thead>
<tr>
<th>Specification</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: X,Y,Z (Gauss)</td>
<td>+/- 8</td>
</tr>
<tr>
<td>Resolution (mG)</td>
<td>&lt; 0.3</td>
</tr>
<tr>
<td>Noise Density (mG/Hz)</td>
<td>&lt; 0.25</td>
</tr>
<tr>
<td>Bandwidth (Hz)</td>
<td>5</td>
</tr>
</tbody>
</table>

Note: This product has been developed and is sold exclusively for commercial applications. Military usage is prohibited by the manufacturer.

Note: This product is subject to US Department of Commerce Export Controls ECCN 7A994, ECCN6A996. Any use of this product for nuclear, chemical or biological weapons, or weapons research, or for any use in missiles, rockets, and/or UAV’s of 300km or greater range, or any other activity prohibited by the Export Administration Regulations, is expressly prohibited without the written consent and without obtaining appropriate US export license(s) when required by US law. Diversion contrary to U.S. law is prohibited. Specifications are subject to change without notice.

### 7.1.2 Environmental, Electrical, and Physical Specifications

#### ENVIRONMENT

<table>
<thead>
<tr>
<th>Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Temperature (°C)</td>
<td>-40 to 105</td>
</tr>
<tr>
<td>Non-Operating Temperature (°C)</td>
<td>-65 to 150</td>
</tr>
<tr>
<td>Enclosure</td>
<td></td>
</tr>
</tbody>
</table>

#### ELECTRICAL

<table>
<thead>
<tr>
<th>Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Voltage (VDC)</td>
<td>3.0 - 5.5</td>
</tr>
<tr>
<td>Power Consumption (mW)</td>
<td>&lt; 250</td>
</tr>
<tr>
<td>Digital Interface</td>
<td>SPI or UART</td>
</tr>
<tr>
<td>Output Date Rate</td>
<td>up to 200Hz (SPI) up to 100Hz (UART)</td>
</tr>
<tr>
<td>Input Clock Sync</td>
<td>1KHz pulse (Configurable)</td>
</tr>
</tbody>
</table>

#### PHYSICAL

<table>
<thead>
<tr>
<th>Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (mm)</td>
<td>24.15 x 37.70 x 9.50</td>
</tr>
<tr>
<td>Weight (gm)</td>
<td>&lt; 17</td>
</tr>
<tr>
<td>Interface Connector</td>
<td>20-Pin (10x2) 1.0mm pitch header</td>
</tr>
</tbody>
</table>

### 7.1. OpenIMU300ZI - EZ Embed Automotive Module
7.1.3 Module Mechanical Drawing

Dimensions are in mm

7.1.4 SPI and UART

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:

| 3 UART Mode          | • User UART
|                      | • GPS/External Sensor UART
|                      | • Debug UART
| UART + SPI Mode      | • User SPI Port
|                      | • GPS/Debug UART

7.1.5 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.

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
Fig. 1: OpenIMU300ZI Module Mechanical Drawing
<table>
<thead>
<tr>
<th>Pin</th>
<th>Main Function</th>
<th>Alternative Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GPIO ( IO3 )</td>
<td>Output by default</td>
</tr>
<tr>
<td>2</td>
<td>Synchronization Input</td>
<td>GPS 1PPS Input</td>
</tr>
<tr>
<td>3</td>
<td>User UART TX (Output) (Serial Channel 0)</td>
<td>SPI Clock (SCLK) Input</td>
</tr>
<tr>
<td>4</td>
<td>User UART RX (Input) (Serial Channel 0)</td>
<td>SPI Data (MISO) Output</td>
</tr>
<tr>
<td>5</td>
<td>UART1 TX (Output) (Serial Channel 1)</td>
<td>SPI Data (MOSI) Input</td>
</tr>
<tr>
<td>6</td>
<td>UART1 RX (Input) (Serial Channel 1)</td>
<td>SPI Chip Select (SS) Input</td>
</tr>
<tr>
<td>7</td>
<td>SPI/UART Interface Selector</td>
<td>Data Ready (SPI) Active edge falling</td>
</tr>
<tr>
<td>8</td>
<td>External Reset (NRST)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>GPIO ( IO2 )</td>
<td>Output by default</td>
</tr>
<tr>
<td>10</td>
<td>Power VIN (3-5 VDC)</td>
<td>Power VIN (3-5 VDC)</td>
</tr>
<tr>
<td>11</td>
<td>Power VIN (3-5 VDC)</td>
<td>Power VIN (3-5 VDC)</td>
</tr>
<tr>
<td>12</td>
<td>Power VIN (3-5 VDC)</td>
<td>Power VIN (3-5 VDC)</td>
</tr>
<tr>
<td>13</td>
<td>Power GND</td>
<td>Power GND</td>
</tr>
<tr>
<td>14</td>
<td>Power GND</td>
<td>Power GND</td>
</tr>
<tr>
<td>15</td>
<td>Power GND</td>
<td>Power GND</td>
</tr>
<tr>
<td>16</td>
<td>SWDIO (SWD debug interface)</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>UART2 TX (Serial Channel 2)</td>
<td>Debug interface GPS</td>
</tr>
<tr>
<td>18</td>
<td>SWCLK (SWD debug interface)</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>UART2 RX (Serial Channel 2)</td>
<td>Debug Interface GPS</td>
</tr>
<tr>
<td>20</td>
<td>Reference voltage for SWD debug interface</td>
<td></td>
</tr>
</tbody>
</table>
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.

7.1.6 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.

7.2 OpenIMU300RI - Robust Industrial CAN Module

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, -40°C to +85°C
  - 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. 2: OpenIMU300RI Default Coordinate Frame

+Z Axis goes out the bottom of the unit

Roll

Pitch

+X Axis

+Y Axis
• 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

7.2.1 Internal Sensors

ANGULAR RATE

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: X,Y,Z (°/Sec)</td>
<td>&gt; 400°/sec</td>
</tr>
<tr>
<td>Bias Instability (°/Hr)</td>
<td>&lt; 6°/hr</td>
</tr>
<tr>
<td>Bias Stability Over Temperature (°/Sec)</td>
<td>&lt; 0.5°/sec</td>
</tr>
<tr>
<td>Resolution (°/Sec)</td>
<td>&lt; 0.02°/sec</td>
</tr>
<tr>
<td>Scale Factor Accuracy (%)</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>Non-Linearity (%)</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>Angle Random Walk (°/hr)</td>
<td>&lt; 0.3°/hr</td>
</tr>
<tr>
<td>Bandwidth (Hz)</td>
<td>5-50Hz (Configurable)</td>
</tr>
</tbody>
</table>

Notes: 1 Allen Variance Curve, Constant Temperature, 2 1-Sigma Error, 3 RMS Error over Temperature

ACCELERATION

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: X,Y,Z (g)</td>
<td>&gt; 8 g</td>
</tr>
<tr>
<td>Bias Instability (g)</td>
<td>&lt; 20 mg</td>
</tr>
<tr>
<td>Bias Stability Over Temperature (mg)</td>
<td>&lt; 5 mg</td>
</tr>
<tr>
<td>Resolution (mg)</td>
<td>&lt; 0.5 mg</td>
</tr>
<tr>
<td>Scale Factor Accuracy (%)</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>Non-Linearity (%)</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>Velocity Random Walk (m/s/hr)</td>
<td>&lt; 0.05 m/s/hr</td>
</tr>
<tr>
<td>Bandwidth (Hz)</td>
<td>5-50Hz (Configurable)</td>
</tr>
</tbody>
</table>

Notes: 1 Allen Variance Curve, Constant Temperature, 2 1-Sigma Error, 3 RMS Error over Temperature

MAGNETIC FIELD

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: X,Y,Z (Gauss)</td>
<td>+/- 8 Gauss</td>
</tr>
<tr>
<td>Resolution (mG)</td>
<td>&lt; 0.3 mG</td>
</tr>
<tr>
<td>Noise Density (mG/Hz)</td>
<td>&lt; 0.25 mG/Hz</td>
</tr>
<tr>
<td>Bandwidth (Hz)</td>
<td>5 Hz</td>
</tr>
</tbody>
</table>
Note: This product has been developed and is sold exclusively for commercial applications. Military usage is prohibited by the manufacturer.

Note: This product is subject to US Department of Commerce Export Controls ECCN 7A994, ECCN6A996. Any use of this product for nuclear, chemical or biological weapons, or weapons research, or for any use in missiles, rockets, and/or UAV’s of 300km or greater range, or any other activity prohibited by the Export Administration Regulations, is expressly prohibited without the written consent and without obtaining appropriate US export license(s) when required by US law. Diversion contrary to U.S. law is prohibited. Specifications are subject to change without notice.

7.2.2 Preliminary Specifications

![Performance Specification](image)

![Electrical Specifications](image)

![Physical Specifications](image)

![Environmental Specifications](image)

7.2.3 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.

7.2.4 CAN and UART

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:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART Mode</td>
<td>Typically used during early development</td>
</tr>
<tr>
<td></td>
<td>Single UART for all messages, debug output, and firmware update</td>
</tr>
<tr>
<td>CAN + UART Mode</td>
<td>Typically used during late development</td>
</tr>
<tr>
<td></td>
<td>Uses CAN Port for messages and firmware update</td>
</tr>
<tr>
<td></td>
<td>Single UART for all messages, debug output, and firmware update</td>
</tr>
<tr>
<td>CAN Mode</td>
<td>Typically used for production</td>
</tr>
<tr>
<td></td>
<td>Uses CAN Port for messages and firmware update</td>
</tr>
</tbody>
</table>

7.2.5 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.

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.

Note: Power is applied to the OpenIMU300RI on pin 6. Pin 3 is ground. The OpenIMU300RI accepts an unregulated
Fig. 4: OpenIMU300RI Connector

Fig. 5: OpenIMU300RI Connector Pinout

<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CAN H</td>
</tr>
<tr>
<td>2</td>
<td>CAN L</td>
</tr>
<tr>
<td>3</td>
<td>Ground</td>
</tr>
<tr>
<td>4</td>
<td>RS232 RX</td>
</tr>
<tr>
<td>5</td>
<td>RS232 TX</td>
</tr>
<tr>
<td>6</td>
<td>Power</td>
</tr>
</tbody>
</table>

Fig. 6: OpenIMU300RI Connector Pinout
4.9 to 32 VDC input. It is reverse polarity and ESD protected internally.

7.2.6 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.

7.3 OpenIMU330BI - EZ Embed Automotive Module

The following image shows the OpenIMU330BI unit.
The OpenIMU330BI *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.

OpenIMU330BI 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. The main feature of OpenIMU330BI is triple redundancy for each inertial sensor.

- Integrated triple redundant 3-Axis Angular Rate Sensors
- Integrated triple redundant 3-Axis Accelerometer
- 80MHz STM32 M4 CPU with FPU
- SPI / UART Interfaces
• Sensors Update Rate to 800Hz
• Synchronization Input
• In-System Upgrade
• Ultra Small Size: TBD
• Wide Temp Range -40 to 85 °C
• High Reliability > 50,000hr MTBF
• Low power

7.3.1 OpenIMU330BI Sensors

ANGULAR RATE

<table>
<thead>
<tr>
<th>Specification</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: X,Y,Z (°/Sec)</td>
<td>&gt; 400</td>
</tr>
<tr>
<td>Bias Instability (°/Hr)</td>
<td>&lt; 6</td>
</tr>
<tr>
<td>Bias Stability Over Temperature (°/Sec)</td>
<td>&lt; 0.5</td>
</tr>
<tr>
<td>Resolution (°/Sec)</td>
<td>&lt; 0.02</td>
</tr>
<tr>
<td>Scale Factor Accuracy (%)</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>Non-Linearity (%)</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>Angle Random Walk (°/hr)</td>
<td>&lt; 0.3</td>
</tr>
<tr>
<td>Bandwidth (Hz)</td>
<td>5-50Hz (Configurable)</td>
</tr>
</tbody>
</table>

Notes: 1 Allen Variance Curve, Constant Temperature, 2 1-Sigma Error, 3 RMS Error over Temperature

ACCELERATION

<table>
<thead>
<tr>
<th>Specification</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: X,Y,Z (g)</td>
<td>&gt; 8</td>
</tr>
<tr>
<td>Bias Instability (g)</td>
<td>&lt; 20</td>
</tr>
<tr>
<td>Bias Stability Over Temperature (mg)</td>
<td>&lt; 5</td>
</tr>
<tr>
<td>Resolution (mg)</td>
<td>&lt; 0.5</td>
</tr>
<tr>
<td>Scale Factor Accuracy (%)</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>Non-Linearity (%)</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>Velocity Random Walk (m/s/hr)</td>
<td>&lt; 0.05</td>
</tr>
<tr>
<td>Bandwidth (Hz)</td>
<td>5-50Hz (Configurable)</td>
</tr>
</tbody>
</table>

Notes: 1 Allen Variance Curve, Constant Temperature, 2 1-Sigma Error, 3 RMS Error over Temperature
Note: This product has been developed and is sold exclusively for commercial applications. Military usage is prohibited by the manufacturer.

Note: This product is subject to US Department of Commerce Export Controls ECCN 7A994, ECCN6A996. Any use of this product for nuclear, chemical or biological weapons, or weapons research, or for any use in missiles, rockets, and/or UAV’s of 300km or greater range, or any other activity prohibited by the Export Administration Regulations, is expressly prohibited without the written consent and without obtaining appropriate US export license(s) when required by US law. Diversion contrary to U.S. law is prohibited. Specifications are subject to change without notice.

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

#### ENVIRONMENT

<table>
<thead>
<tr>
<th>Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Temperature (°C)</td>
<td>-40 to 105</td>
</tr>
<tr>
<td>Non-Operating Temperature (°C)</td>
<td>-65 to 150</td>
</tr>
<tr>
<td>Enclosure</td>
<td></td>
</tr>
</tbody>
</table>

#### ELECTRICAL

<table>
<thead>
<tr>
<th>Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Voltage (VDC)</td>
<td>3.0 - 5.5</td>
</tr>
<tr>
<td>Power Consumption (mW)</td>
<td>&lt; 250</td>
</tr>
<tr>
<td>Digital Interface</td>
<td>SPI or UART</td>
</tr>
<tr>
<td>Output Date Rate</td>
<td>up to 200Hz (SPI)</td>
</tr>
<tr>
<td></td>
<td>up to 100Hz (UART)</td>
</tr>
<tr>
<td>Input Clock Sync</td>
<td>1KHz pulse (Configurable)</td>
</tr>
</tbody>
</table>

#### PHYSICAL

<table>
<thead>
<tr>
<th>Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (mm)</td>
<td>TBD</td>
</tr>
<tr>
<td>Weight (gm)</td>
<td>TBD</td>
</tr>
<tr>
<td>Interface Connector</td>
<td>TBD</td>
</tr>
</tbody>
</table>

### 7.3.3 OpenIMU330BI
Module Mechanical Drawing

Dimensions are in mm
Fig. 7: OpenIMU330BI Module Mechanical Drawing
### 7.3.4 OpenIMU330BI

#### SPI and UART

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

Typical configurations include:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 UART Mode</td>
<td>• User UART</td>
</tr>
<tr>
<td></td>
<td>• Debug UART</td>
</tr>
<tr>
<td></td>
<td>• Debug UART</td>
</tr>
<tr>
<td>UART + SPI Mode</td>
<td>• User SPI Port</td>
</tr>
<tr>
<td></td>
<td>• Debug UART</td>
</tr>
</tbody>
</table>

#### 7.3.5 OpenIMU330BI

**Unit package Pinout**

![Pinout Diagram](image-url)
7.3.6 ARM Cortex M4 CPU

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

- FPU
- 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
Part IV

Evaluation Kits
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.
The following pages cover:

- Detailed Overview
- Eval Board Mechanical Drawing
- Eval Board Schematic

### 8.1 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 via 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

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

8.1. Overview
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.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Main Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Vref</td>
</tr>
<tr>
<td>2, 4, 6, 8, 10, 12, 14, 16, 18, 20</td>
<td>GND</td>
</tr>
<tr>
<td>7</td>
<td>SWDIO</td>
</tr>
<tr>
<td>9</td>
<td>SWCLK</td>
</tr>
<tr>
<td>15</td>
<td>nRST</td>
</tr>
<tr>
<td>19</td>
<td>3.3V from debugger</td>
</tr>
</tbody>
</table>

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.

7. OpenIMU Evaluation Kit Important Notice

This evaluation kit is intended for use 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.
8.2 OpenIMU300ZI EVK Mechanical Drawing

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

Mechanical Drawing download link
8.3 EVB Schematic

[Diagram of EVB Schematic]

Schematic download
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.
OpenIMU300RI Evaluation Unit
installed on test fixture with JTAG connector

OpenIMU300RI Evaluation Kit

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

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Unit Connector</th>
<th>RS232 Connector</th>
<th>CAN Connector</th>
<th>Power Wires</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>3</td>
<td>5</td>
<td></td>
<td>Black</td>
</tr>
<tr>
<td>VIN</td>
<td>6</td>
<td></td>
<td></td>
<td>Red</td>
</tr>
<tr>
<td>RS232 TX</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS232 RX</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN H</td>
<td>1</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>CAN L</td>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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.
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.
The following pages cover:

- Detailed Overview
- Eval Board Mechanical Drawing
- Eval Board Schematic

10.1 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
<table>
<thead>
<tr>
<th>Pin</th>
<th>Main Function</th>
<th>Alternative Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Power GND</td>
<td>Power GND</td>
</tr>
<tr>
<td>2</td>
<td>Power GND</td>
<td>Power GND</td>
</tr>
<tr>
<td>3</td>
<td>Serial Channel 1 RX (Input)</td>
<td>SPI Chip Select (SS) (Input)</td>
</tr>
<tr>
<td>4</td>
<td>IMU Data Ready (SPI interface Mode)</td>
<td>GPIO (UART interface mode)</td>
</tr>
<tr>
<td>5</td>
<td>User UART TX (Serial Channel 0) (Output)</td>
<td>SPI Clock (SCK) (Output)</td>
</tr>
<tr>
<td>6</td>
<td>Synchronization Input</td>
<td>1PPS Input from GPS</td>
</tr>
<tr>
<td>7</td>
<td>User UART TX (Serial Channel 0) (Output)</td>
<td>SPI Data (MOSI) (Input)</td>
</tr>
<tr>
<td>8</td>
<td>External Reset (NRST)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>SPI Data (MISO) (Output)</td>
</tr>
<tr>
<td>10</td>
<td>GPIO Output (IO2)</td>
<td>GPIO Input</td>
</tr>
<tr>
<td>11</td>
<td>Power VIN 5 VDC</td>
<td>Power VIN 5 VDC</td>
</tr>
<tr>
<td>12</td>
<td>GPIO Output (IO3)</td>
<td>GPIO Input</td>
</tr>
<tr>
<td>17</td>
<td>Debug UART TX</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Debug UART RX</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Main Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Vref</td>
</tr>
<tr>
<td>2, 4, 6, 8, 10, 12, 14, 16, 18, 20</td>
<td>GND</td>
</tr>
<tr>
<td>7</td>
<td>SWDIO</td>
</tr>
<tr>
<td>9</td>
<td>SWCLK</td>
</tr>
<tr>
<td>15</td>
<td>nRST</td>
</tr>
<tr>
<td>19</td>
<td>3.3V from debugger</td>
</tr>
</tbody>
</table>

3.7 USB connector (J3)
USB connector used for powering up the IMU and evaluation board. Also, it is 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.

7. 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**...

(continues on next 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.

10.2 OpenIMU330BI EVK Mechanical Drawing

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

Mechanical Drawing download link
10.3 OpenIMU330BI EVB Schematic

Schematic download
Part V

Software Development
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*.

**11.1 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.

**11.1.1 FreeRTOS & Board Support Package**

**FreeRTOS**

The applications for all OpenIMU units use the FreeRTOS Real-Time Operating System ([https://www.freertos.org](https://www.freertos.org)). 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](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**

### 11.1.2 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 straightforward 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 filters 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.
11.1.3 Default Data Acquisition Functions

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:

```c
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>
<thead>
<tr>
<th>Getter Function</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetAccelData_g()</code></td>
<td>Obtain accelerometer data</td>
<td>[g]</td>
</tr>
<tr>
<td><code>GetAccelData_mPerSecSq()</code></td>
<td>Obtain accelerometer data</td>
<td>[m/s²]</td>
</tr>
<tr>
<td><code>GetRateData_radPerSec()</code></td>
<td>Obtain rate-sensor data</td>
<td>[r/s]</td>
</tr>
<tr>
<td><code>GetRateData_degPerSec()</code></td>
<td>Obtain rate-sensor data</td>
<td>[°/s]</td>
</tr>
<tr>
<td><code>GetMagData_G()</code></td>
<td>Obtain magnetometer data</td>
<td>[G]</td>
</tr>
<tr>
<td><code>GetBoardTempData()</code></td>
<td>Obtain temperature data</td>
<td>[°C]</td>
</tr>
</tbody>
</table>

**Note:** Most inertial algorithm development will use \([m/s^2]\), \([r/s]\), 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.

```c
// IMU data structure
typedef struct {
    // Timer output counter
    uint32_t timerCntr, dTimerCntr;
    // Algorithm states

    // (continues on next page)
```
double accel_g[3];
double rate_degPerSec[3];
double mag_G[3];
double temp_C;
} IMUDataStruct;

extern IMUDataStruct gIMU;

11.1.4 Default Message Functions

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 [°/s]
5. Magnetometer readings, in [G]
6. Board temperature, in [°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>
<thead>
<tr>
<th>Message Component</th>
<th>Description</th>
<th>Number of Variables</th>
<th>Total Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer counter</td>
<td>uint32_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time variable</td>
<td>double</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accelerometer Readings (3 axis)</td>
<td>float</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>Rate-Sensor Readings (3 axis)</td>
<td>float</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>Magnetometer Readings (3 axis)</td>
<td>float</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>Board-Temperature Readings (3 axis)</td>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

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:

```c
// 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

```c
#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:

```c
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:

```c
case USR_OUT_SCALED1: // packet with arbitrary data
    _outputPacketType = type;
    _userPayloadLen = USR_OUT_SCALED1_PAYLOAD_LEN;
(continues on next page)
```
and the packet-code must be added to the list of user output packets, `userOutputPackets`.

```c
usr_packet_t userOutputPackets[] = {
    {USR_OUT_NONE, {0x00, 0x00}},
    {USR_OUT_TEST, "zT"},
    {USR_OUT_DATA1, "z1"},
    {USR_OUT_DATA2, "z2"},
    {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`:

```c
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”.

---

11.1. OpenIMU Core Details 92
If present, the message is being generated; whether the message is populated correctly requires the use of additional tools.

![Tera Term - [disconnected] VT](image)

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.

### 11.1.5 Default Serial Debugging Functions

#### Contents

- Generating Debug Messages
- Compile and Test

**Generating Debug Messages**

Creating the Message:
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:

```c
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:

```c
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](image)

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.

### 11.1.6 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-openuim\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
<240, 142560>
Success
Restarting app ...
autoconnected
Connected ....OpenIMU3002A5020-3885-01 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.

## 11.1.7 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": "s1",
"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"}]}
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

11.1.8 Data Capture Functions Supporting the Aceinna Navigation Studio
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
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:

![Server-Connection Message at the Terminal Prompt](image)

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](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

11.1. OpenIMU Core Details
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:

![Plot of IMU Accelerometer Data](image)

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 \( \frac{g}{s} \))
- Rate-Sensor data (in \( \frac{°}{s} \))
- Magnetometer data (in \( G \))
- Board-Temperature data (in \( °C \))

The following figure shows the contents of the captured data file, indicating that all selected data are saved as intended.
11.2 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 \( [°/s] \)
  - Magnetic-field readings in \( [G] \)
  - Sensor temperature readings in \( [°C] \)

11.3 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\(^1\) consisting of the following:
  - A relative time measurement (both integer and decimal values)
  - Acceleration readings in \([g]\)
  - Rate-sensor readings in \([°/s]\)
  - Magnetic-field readings in \([G]\)
  - Sensor temperature readings in \([°C]\)

**11.4 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

\(^1\) The output message is the same as for the IMU application, but tailored by the Leveler algorithm
11.5 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

11.6 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.
11.6.1 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.

![Fig. 13: Base folder from VSCode CAN J1939 Workspace](tutorial/CAN/../../media/OpenIMU300RI-CAN_J1939_Project.png)

11.6.2 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.

11.6.3 Messages in the CAN J1939 Application Example

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. 14: **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

<table>
<thead>
<tr>
<th>Request</th>
<th>PF (dec)</th>
<th>PS (dec)</th>
<th>PGN</th>
<th>Payload Length (bytes)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save Configuration</td>
<td>255</td>
<td>81</td>
<td>65361</td>
<td>3</td>
<td>Save all configuration data to non-volatile memory</td>
</tr>
<tr>
<td>Reset Algorithm</td>
<td>255</td>
<td>80</td>
<td>65360</td>
<td>3</td>
<td>Reset algorithm to initial conditions</td>
</tr>
<tr>
<td>Mag Alignment</td>
<td>255</td>
<td>94</td>
<td>65374</td>
<td>2</td>
<td>Mag alignment and status request</td>
</tr>
<tr>
<td>Set Packet Rate Divider</td>
<td>255</td>
<td>85</td>
<td>65365</td>
<td>2</td>
<td>Set rate dividers to increase/decrease rate packets are set</td>
</tr>
<tr>
<td>Set Data Packet Type(s)</td>
<td>255</td>
<td>86</td>
<td>65366</td>
<td>2</td>
<td>Select packet types to be sent periodically</td>
</tr>
<tr>
<td>Set Digital Filters Cutoff Frequencies</td>
<td>255</td>
<td>87</td>
<td>65367</td>
<td>3</td>
<td>Set LPF cutoff frequency for rate sensors and accelerometers</td>
</tr>
<tr>
<td>Set Orientation</td>
<td>255</td>
<td>88</td>
<td>65368</td>
<td>3</td>
<td>Set unit orientation</td>
</tr>
<tr>
<td>Set Bank of PS Numbers for Bank0</td>
<td>255</td>
<td>240</td>
<td>65520</td>
<td>8</td>
<td>Reconfigure PS numbers for set requests</td>
</tr>
<tr>
<td>Set Bank of PS Numbers for Bank1</td>
<td>255</td>
<td>241</td>
<td>65521</td>
<td>8</td>
<td>Reconfigure PS numbers for set requests</td>
</tr>
</tbody>
</table>
Note: PS values for all but the “Set Bank of PS Numbers for Bank0/Bank1” Set Commands can be changed by 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

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description/Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Type: 0 = Request, 1 = Response</td>
</tr>
<tr>
<td>1</td>
<td>Destination Address</td>
</tr>
<tr>
<td>2</td>
<td>Response: 0 = Fail, 1 = Succeed</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description/Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Type: 0 = Request, 1 = Response</td>
</tr>
<tr>
<td>1</td>
<td>Destination Address</td>
</tr>
<tr>
<td>2</td>
<td>Response: 0 = Fail, 1 = Succeed</td>
</tr>
</tbody>
</table>

Mag Alignment (INS Application Example)

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

Table 6: Mag Alignment Request Payload Fields

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description/Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Destination Address</td>
</tr>
<tr>
<td>1</td>
<td>Commands:</td>
</tr>
<tr>
<td></td>
<td>0 - Status Request</td>
</tr>
<tr>
<td></td>
<td>1 - Start Alignment</td>
</tr>
<tr>
<td></td>
<td>5 - Confirm and Save</td>
</tr>
</tbody>
</table>
### Table 7: Payload Fields of 64 bit Response

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 0:7</td>
<td>Command</td>
<td>0 - Status request, 1 - Start alignment</td>
</tr>
</tbody>
</table>
| 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

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
<th>Byte Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Destination Address</td>
<td>Unique</td>
</tr>
<tr>
<td>1</td>
<td>Packet Divider Value</td>
<td><strong>Byte Value - Packet Broadcast Rate (Hz)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - Quiet Mode - no broadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - 100 (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 - 50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 - 25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 - 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 - 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20 - 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25 - 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50 - 2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
<th>Byte Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Destination Address</td>
<td>Unique</td>
</tr>
<tr>
<td>1</td>
<td>Selected Data Packet Type(s) Bitmask (LSB)</td>
<td>Data Packet Type(s) Bitmask: Bit 0 - SSI2 Bit 1 - Angular Rate Bit 2 - Acceleration</td>
</tr>
<tr>
<td>2</td>
<td>Selected Data Packet Type(s) Bitmask (MSB)</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Set Digital Filter Cutoff Frequencies

The following table shows provides descriptions of the response payload.
Table 10: *Digital Filter Cutoff Frequencies Request Payload*

<table>
<thead>
<tr>
<th>Payload Byte</th>
<th>Description/Values</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Destination Address</td>
<td>Unique</td>
</tr>
<tr>
<td>1</td>
<td>Cutoff Frequencies (Hz) for Angular Rate Sensors</td>
<td>0, 2, 5, 10, 25, 40, or 50</td>
</tr>
<tr>
<td>2</td>
<td>Cutoff Frequencies (Hz) for Accelerometer Sensors</td>
<td>0, 2, 5, 10, 25, 40, or 50</td>
</tr>
</tbody>
</table>

**Set Orientation**

The following table shows the payload layout

Table 11: *Set Orientation Payload Layout*

<table>
<thead>
<tr>
<th>Byte</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Destination Address</td>
<td>Unique</td>
</tr>
<tr>
<td>1:2</td>
<td>Orientation Setting</td>
<td>see table below</td>
</tr>
</tbody>
</table>

The following table provides the values and meanings of the payload field bytes:

Table 12: *Set Orientation Field Descriptions*

<table>
<thead>
<tr>
<th>Orientation Field Value</th>
<th>X/Y/Z Axis</th>
<th>Orientation Field Value</th>
<th>X/Y/Z Axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>+Ux +Uy +Uz (default)</td>
<td>0x00C4</td>
<td>+Uz +Uy -Ux</td>
</tr>
<tr>
<td>0x0009</td>
<td>-Ux -Uy +Uz</td>
<td>0x00CD</td>
<td>-Uz -Uy -Ux</td>
</tr>
<tr>
<td>0x0023</td>
<td>-Uy +Ux +Uz</td>
<td>0x00D3</td>
<td>-Uy +Uz -Ux</td>
</tr>
<tr>
<td>0x002A</td>
<td>+Uy -Ux +Uz</td>
<td>0x00DA</td>
<td>+Uy -Uz -Ux</td>
</tr>
<tr>
<td>0x0041</td>
<td>-Ux +Uy -Uz</td>
<td>0x0111</td>
<td>-Ux +Uz +Uy</td>
</tr>
<tr>
<td>0x0048</td>
<td>+Uy -Ux -Uz</td>
<td>0x0118</td>
<td>+Ux -Uz +Uy</td>
</tr>
<tr>
<td>0x0062</td>
<td>+Uy +Ux -Uz</td>
<td>0x0124</td>
<td>+Uz +Ux +Uy</td>
</tr>
<tr>
<td>0x006B</td>
<td>-Uy -Ux -Uz</td>
<td>0x012D</td>
<td>-Uz -Ux +Uy</td>
</tr>
<tr>
<td>0x0085</td>
<td>-Uz +Uy +Ux</td>
<td>0x0150</td>
<td>+Ux +Uz -Uy</td>
</tr>
<tr>
<td>0x008C</td>
<td>+Uz -Uy +Ux</td>
<td>0x0159</td>
<td>-Ux -Uz -Uy</td>
</tr>
<tr>
<td>0x0092</td>
<td>+Uy +Uz +Ux</td>
<td>0x0165</td>
<td>-Uz +Ux -Uy</td>
</tr>
<tr>
<td>0x009B</td>
<td>-Uy -Uz +Ux</td>
<td>0x016C</td>
<td>+Uz -Ux -Uy</td>
</tr>
</tbody>
</table>
Set Bank of PS Numbers

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

<table>
<thead>
<tr>
<th>Byte</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Destination Address</td>
</tr>
<tr>
<td>1</td>
<td>Reset Algorithm PS number</td>
</tr>
<tr>
<td>2</td>
<td>Save Configuration PS number</td>
</tr>
<tr>
<td>3</td>
<td>Status Request PS number</td>
</tr>
<tr>
<td>4</td>
<td>Mag Align Command PS number</td>
</tr>
<tr>
<td>5-7</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 13: Set Bank of PS Numbers for Bank0 Payload

<table>
<thead>
<tr>
<th>Byte</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Destination Address</td>
</tr>
<tr>
<td>1</td>
<td>Set Packet Rate PS number</td>
</tr>
<tr>
<td>2</td>
<td>Set Packet Type(s) PS number</td>
</tr>
<tr>
<td>3</td>
<td>Set Digital Filer Cutoff Frequencies PS number</td>
</tr>
<tr>
<td>4</td>
<td>Set Orientation PS Number</td>
</tr>
<tr>
<td>5</td>
<td>Set User Behavior PS Number</td>
</tr>
<tr>
<td>6-7</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 14: Set Bank of PS Numbers for Bank1 Payload

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 : Destination Address,
- DLC : 3,
- Priority: 6,
- PGN : 59904.

Table 15: Request Payload

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>PF of requested parameter</td>
</tr>
<tr>
<td>2</td>
<td>PS of requested parameter</td>
</tr>
</tbody>
</table>
In the table below, the list of parameters which can be requested from ECU, including their PF, PS, and payload length of response messages.

### Table 16: List of ECU parameters available for Requests

<table>
<thead>
<tr>
<th>Parameter</th>
<th>PF (dec)</th>
<th>PS (dec)</th>
<th>Payload Length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Version</td>
<td>254</td>
<td>218</td>
<td>5</td>
</tr>
<tr>
<td>ECU ID</td>
<td>253</td>
<td>197</td>
<td>8</td>
</tr>
<tr>
<td>Packet Rate</td>
<td>255</td>
<td>85</td>
<td>2</td>
</tr>
<tr>
<td>Packet Type</td>
<td>255</td>
<td>86</td>
<td>3</td>
</tr>
<tr>
<td>Digital Cutoff Frequency</td>
<td>255</td>
<td>87</td>
<td>3</td>
</tr>
<tr>
<td>Orientation</td>
<td>255</td>
<td>88</td>
<td>3</td>
</tr>
</tbody>
</table>

**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, the values of PF and PS field in extended headers have the same PF+PS values as requested.

### Responses to Get Requests

The following table describes the payloads for responses to Get Requests.

### Table 17: Software Version Response Payload

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Major Version Number</td>
</tr>
<tr>
<td>1</td>
<td>Minor Version Number</td>
</tr>
<tr>
<td>2</td>
<td>Patch Number</td>
</tr>
<tr>
<td>3</td>
<td>Stage Number</td>
</tr>
<tr>
<td>4</td>
<td>Build Number</td>
</tr>
</tbody>
</table>
Table 18: *ECU ID 64 Bit Response Payload*

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

Table 19: *Packet Rate Response Payload*

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Source Address</td>
</tr>
<tr>
<td>1</td>
<td>Output Data Rate</td>
</tr>
</tbody>
</table>

Table 20: *Packet Type Response Payload*

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Source Address</td>
</tr>
<tr>
<td>1</td>
<td>Packet Types Bitmask (LSB)</td>
</tr>
<tr>
<td>2</td>
<td>Packet Types Bitmask (MSB)</td>
</tr>
</tbody>
</table>

Table 21: *Digital Cutoff Frequency Response Payload*

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Source Address</td>
</tr>
<tr>
<td>1</td>
<td>Acceleration Cutoff</td>
</tr>
<tr>
<td>2</td>
<td>Angular Rate Cutoff</td>
</tr>
</tbody>
</table>

Table 22: *Orientation Response Payload*

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Source Address</td>
</tr>
<tr>
<td>1</td>
<td>First Byte of Orientation Value</td>
</tr>
<tr>
<td>2</td>
<td>Second Byte of Orientation Value</td>
</tr>
</tbody>
</table>

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 23: Data Messages

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

Slope Sensor Information - Type 2 (SSI2) Data Packet

The following table describes the SSI2 Data Packet Payload:
Table 24: SSI2 Data Packet Payload

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

Angular Rate Data Packet

The following table describes the Angular Rate Data Packet:

Table 25: Angular Rate Data Packet Payload

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

Acceleration Data Packet

The following table describes the Acceleration Data Packet:

Table 26: Acceleration Data Packet Payload

<table>
<thead>
<tr>
<th>Byte Number</th>
<th>Parameter</th>
<th>Range</th>
<th>Resolution</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:1</td>
<td>Acceleration X</td>
<td>-320 to 320/55 m/s**2</td>
<td>0.01 m/s**2/bit</td>
<td>-320 m/s**2</td>
</tr>
<tr>
<td>2:3</td>
<td>Acceleration Y</td>
<td>-320 to 320/55 m/s**2</td>
<td>0.01 m/s**2/bit</td>
<td>-320 m/s**2</td>
</tr>
<tr>
<td>4:5</td>
<td>Acceleration Z</td>
<td>-320 to 320/55 m/s**2</td>
<td>0.01 m/s**2/bit</td>
<td>-320 m/s**2</td>
</tr>
<tr>
<td>6:7</td>
<td>FoM,Latency</td>
<td>Ignore</td>
<td>Ignore</td>
<td>Ignore</td>
</tr>
</tbody>
</table>
Magnetometer Data Packet

The following table describes the Magnetometer Data Packet:

Table 27: Magnetometer Data Packet Payload

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

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

### 12.1 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

### 12.2 FreeRTOS

The applications for all OpenIMU units use the FreeRTOS Real-Time Operating System ([FreeRTOS Site](https://www.freertos.org)). 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

### 12.3 Sampling and Filtering Modules

To Be Provided

### 12.4 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:
### 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:

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

#### 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 = 0xFFFF
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

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Code (U2)</th>
<th>Payload Length (U1)</th>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interactive Messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pG</td>
<td>0x7047</td>
<td>0</td>
<td>Ping</td>
<td>Input/Reply Message</td>
</tr>
<tr>
<td>uC</td>
<td>0x7543</td>
<td>N (up to 248)</td>
<td>Update Config Command/Response</td>
<td>Input/Reply Message</td>
</tr>
<tr>
<td>uP</td>
<td>0x7550</td>
<td>12</td>
<td>Update Parameter Command/Response</td>
<td>Input/Reply Message</td>
</tr>
<tr>
<td>uA</td>
<td>0x7541</td>
<td>N (up to 240)</td>
<td>Update All Command/Response</td>
<td>Input/Reply Message</td>
</tr>
<tr>
<td>sC</td>
<td>0x7343</td>
<td>0</td>
<td>Save Config Command/Response</td>
<td>Input/Reply Message</td>
</tr>
<tr>
<td>rD</td>
<td>0x7244</td>
<td>0</td>
<td>Restore Defaults Command/Response</td>
<td>Input/Reply Message</td>
</tr>
<tr>
<td>sC</td>
<td>0x7343</td>
<td>0</td>
<td>Save Config Command/Response</td>
<td>Input/Reply Message</td>
</tr>
</tbody>
</table>

12.4. OpenIMU UART Messaging Framework
5. OpenIMU Interactive Messages

5.1 User Ping Command

<table>
<thead>
<tr>
<th>Ping (‘pG’ = 0x7047)</th>
<th>Preamble</th>
<th>Packet Code</th>
<th>Length</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7047</td>
<td>0</td>
<td>&lt;CRC (U2)&gt;</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Ping (‘pG’ = 0x7047)</th>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7047</td>
<td>N</td>
<td></td>
<td>Unit Model and Serial Number &lt;S&gt; (string)</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

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

5.2 Update Config Command

<table>
<thead>
<tr>
<th>‘uC’ = 0x7543</th>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7543</td>
<td>8+8*N</td>
<td>N Parameters</td>
<td>&lt;CRC (U2)&gt;</td>
<td></td>
</tr>
</tbody>
</table>

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
OpenIMU Documentation

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Number of consecutive parameters to update</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>4</td>
<td>Offset of first parameter in unit config structure</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>8</td>
<td>Parameter Value</td>
<td>U8 or I8 or F8 or double or S8 or A8</td>
<td>LSB First</td>
</tr>
<tr>
<td>8+N*8</td>
<td>Parameter Value</td>
<td>U8 or I8 or F8 or double or S8 or A8</td>
<td>LSB First</td>
</tr>
</tbody>
</table>

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:

```
<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7543</td>
<td>4</td>
<td>Error Code (I4)</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>
```

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)

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7550</td>
<td>12</td>
<td></td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>
```

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
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:

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7550</td>
<td>4</td>
<td>Error Code (I4)</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

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

5.4 Update All Command

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

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Offset of parameter in unit configuration</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>8</td>
<td>Parameter Value</td>
<td>U8 or I8 or F8 or double or S8 or A8</td>
<td>LSB First</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7541</td>
<td>4</td>
<td>Error Code (I4)</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

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

5.5 Save Config Command

<table>
<thead>
<tr>
<th>Save Config (‘sc’ = 0x7343)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preamble</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>0x5555</td>
</tr>
</tbody>
</table>

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 (until new changes will be made).

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

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Code</th>
<th>Length</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7343</td>
<td>0</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

5.6 Get Config Command

<table>
<thead>
<tr>
<th>(‘gC’ = 0x6743)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preamble</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>0x5555</td>
</tr>
</tbody>
</table>

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
<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Number Of consecutive parameters to update</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>4</td>
<td>Offset of first parameter in unit config structure</td>
<td>U4</td>
<td>LSB First</td>
</tr>
</tbody>
</table>

Get Config command will have next response:

```
(‘gC’ = 0x6743)
```

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x6743</td>
<td>8+8*N</td>
<td>N parameters</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

Get Config Response Payload Format in case of success:
<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Number Of consecutive parameters to update</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>4</td>
<td>Offset of first parameter in unit config structure</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>8</td>
<td>Parameter Value</td>
<td>U8 or I8 or F8 or double or S8 or A8</td>
<td>LSB First</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U8 or I8 or F8 or double or S8 or A8</td>
<td>LSB First</td>
</tr>
</tbody>
</table>

Get Config Response Payload Format in case of error:

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x6743</td>
<td>4</td>
<td>Error Code (I4)</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

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

5.7 Get All Command

<table>
<thead>
<tr>
<th>(<em>gA</em> = 0x6741)</th>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x6741</td>
<td>0</td>
<td>&lt;CRC (U2)&gt;</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x6741</td>
<td>8*N</td>
<td>N parameters</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

Get Config Response Payload Format in case of success:
<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Number Of consecutive parameters to update</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>4</td>
<td>Offset of first parameter in unit config structure</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>8</td>
<td>Parameter Value</td>
<td>U8 or I8 or F8 or double or S8 or A8</td>
<td>LSB First</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>8+N*8</td>
<td>Parameter Value</td>
<td>U8 or I8 or F8 or double or S8 or A8</td>
<td>LSB First</td>
</tr>
</tbody>
</table>

Get All Response Payload Format in case of error:

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x6741</td>
<td>4</td>
<td>Error Code (I4)</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

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

5.8 Get Parameter Command

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x6750</td>
<td>4</td>
<td></td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

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
<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Offset of parameter in unit config structure</td>
<td>U4</td>
<td>LSB First</td>
</tr>
</tbody>
</table>

Get Parameter command will have next response:

\[
('gP' = 0x6750)
\]

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x6750</td>
<td>12</td>
<td>parameter</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

Get Parameter response payload format in case of success:

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Offset of parameter in unit config structure</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>4</td>
<td>Parameter Value</td>
<td>U8 or I8 or F8 or double or S8 or A8</td>
<td>LSB First</td>
</tr>
</tbody>
</table>

Get Parameter response payload format in case of error:

\[
('gP' = 0x6750)
\]

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x6750</td>
<td>4</td>
<td>Error Code (I4)</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

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

5.9 Get User Version Command

\[
('gV' = 0x6756)
\]

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Code</th>
<th>Length</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x6756</td>
<td>0</td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

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

12.4. OpenIMU UART Messaging Framework

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)
```

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7a54</td>
<td>4</td>
<td></td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

User Test Message payload has next format:

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Counter</td>
<td>U4</td>
<td>LSB First</td>
</tr>
</tbody>
</table>

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

6.2 User Sensors Data Message

User Sensors Data message has next format:

```
('z1' = 0x7a31)
```

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7a31</td>
<td>40</td>
<td></td>
<td>&lt;CRC (U2)&gt;</td>
</tr>
</tbody>
</table>

User Sensors Data Message payload has next format:
<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>System Timer at the moment of sensors sampling</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>4</td>
<td>Acceleration value for axis X (in G)</td>
<td>F4</td>
<td>LSB First</td>
</tr>
<tr>
<td>8</td>
<td>Acceleration value for axis Y (in G)</td>
<td>F4</td>
<td>LSB First</td>
</tr>
<tr>
<td>12</td>
<td>Acceleration value for axis Z (in G)</td>
<td>F4</td>
<td>LSB First</td>
</tr>
<tr>
<td>16</td>
<td>Rotation speed for axis X (dps)</td>
<td>F4</td>
<td>LSB First</td>
</tr>
<tr>
<td>20</td>
<td>Rotation speed for axis Y (dps)</td>
<td>F4</td>
<td>LSB First</td>
</tr>
<tr>
<td>24</td>
<td>Rotation speed for axis Z (dps)</td>
<td>F4</td>
<td>LSB First</td>
</tr>
<tr>
<td>28</td>
<td>Magnetic field for axis X (G)</td>
<td>F4</td>
<td>LSB First</td>
</tr>
<tr>
<td>32</td>
<td>Magnetic field for axis Y (G)</td>
<td>F4</td>
<td>LSB First</td>
</tr>
<tr>
<td>36</td>
<td>Magnetic field for axis Z (G)</td>
<td>F4</td>
<td>LSB First</td>
</tr>
</tbody>
</table>
6.3 User Arbitrary Data Message

User Arbitrary Data message has next format:

<table>
<thead>
<tr>
<th>Preamble</th>
<th>Packet Type</th>
<th>Length</th>
<th>Payload</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5555</td>
<td>0x7a32</td>
<td>27</td>
<td>&lt;CRC (U2)&gt;</td>
<td></td>
</tr>
</tbody>
</table>

User Arbitrary Data Message payload has next format:

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Name</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>System Timer at the moment of sensors sampling</td>
<td>U4</td>
<td>LSB First</td>
</tr>
<tr>
<td>4</td>
<td>Data of type Byte</td>
<td>U1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Data of type short</td>
<td>I2</td>
<td>LSB First</td>
</tr>
<tr>
<td>7</td>
<td>Data of type int</td>
<td>I4</td>
<td>LSB First</td>
</tr>
<tr>
<td>11</td>
<td>Data of type int64</td>
<td>I8</td>
<td>LSB First</td>
</tr>
<tr>
<td>19</td>
<td>Data of type double</td>
<td>D4</td>
<td>LSB First</td>
</tr>
</tbody>
</table>

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

```c
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.

```c
usr_packet_t userInputPackets[] = {
    {USR_IN_NONE, {0,0}}, // ",
    {USR_IN_PING, "pG"},
    {USR_IN_UPDATE_CONFIG, "uC"},
```

(continues on next page)
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).

```c
    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`

```c
    // 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.

```c
    // 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
    ussr_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
    };
```

(continues on next page)
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.

```c
    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.

```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 = "z1",
    .userPacketRate = 50,
    .lpfAccelFilterFreq = 50,
    .lpfRateFilterFreq = 50,
    .orientation = "+X+Y+Z"
// add default parameter values here, if desired
};
```

5. Done
12.5 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.

The master device must be configured to communicate with the OpenIMU using the following settings:

- Data transferred in 16-bit word-length and MSB-first
- fCLK 2.0 MHz
- CPOL = 1 (clock polarity) and CPHA = 1 (clock phase)

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

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Access Type</th>
<th>Function</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>82,83,84,85,88,89 (0x52 0x53,0x54,0x58,0x59)</td>
<td>r</td>
<td>Unit serial number</td>
<td>BCD format</td>
</tr>
<tr>
<td>86, 87 (0x56, 0x57)</td>
<td>r</td>
<td>Product ID</td>
<td>BCD format</td>
</tr>
<tr>
<td>116 (0x74)</td>
<td>r/w</td>
<td>Unit Orientation MSB</td>
<td>p. 6</td>
</tr>
<tr>
<td>117 (0x75)</td>
<td>r/w</td>
<td>Unit Orientation LSB</td>
<td>p. 6</td>
</tr>
<tr>
<td>56 (0x38)</td>
<td>r/w</td>
<td>LPF Filter Type For Accel</td>
<td>p. 7</td>
</tr>
<tr>
<td>57 (0x39)</td>
<td>r/w</td>
<td>LPF Filter Type For Rate Sensors</td>
<td>p. 7</td>
</tr>
<tr>
<td>62 (0x3E)</td>
<td>r</td>
<td>Sensors Data Request</td>
<td>p. 4</td>
</tr>
</tbody>
</table>

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)).

Next figure illustrates register read over SPI interface:

![Register Read over SPI Interface](image)

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>
<thead>
<tr>
<th>Parameter</th>
<th>Size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>2</td>
<td>Unit Status</td>
</tr>
<tr>
<td>X_Rate</td>
<td>2</td>
<td>Rate Sensor output (X) (200LSB/deg/s)</td>
</tr>
<tr>
<td>Y_Rate</td>
<td>2</td>
<td>Rate Sensor output (Y) (200LSB/deg/s)</td>
</tr>
<tr>
<td>Z_Rate</td>
<td>2</td>
<td>Rate Sensor output (Z) (200LSB/deg/s)</td>
</tr>
<tr>
<td>X_Accel</td>
<td>2</td>
<td>Accel Sensor output(X) (4000LSB/G)</td>
</tr>
<tr>
<td>Y_Accel</td>
<td>2</td>
<td>Accel Sensor output(Y) (4000LSB/G)</td>
</tr>
<tr>
<td>Z_Accel</td>
<td>2</td>
<td>Accel Sensor output(Z) (4000LSB/G)</td>
</tr>
</tbody>
</table>

Read of data block begins when the master requests a read from specific register address (i.e. 0x3E). Next figure 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.

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

Next figure 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 3. OpenIMU Orientation field values

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

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:
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 0x39 (for rate sensors). There are 7 possible low pass filter options (see below). Setting/Writing the field to anything else has no effect.

Table 4. OpenIMU Digital filter choices

<table>
<thead>
<tr>
<th>Value Hex (dec)</th>
<th>Cutoff Frequency</th>
<th>Filter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 (0)</td>
<td>N/A</td>
<td>Unfiltered</td>
</tr>
<tr>
<td>0x30 (48)</td>
<td>50 Hz</td>
<td>Butterworth</td>
</tr>
<tr>
<td>0x90 (144)</td>
<td>40 Hz</td>
<td>Butterworth</td>
</tr>
<tr>
<td>0x80 (128)</td>
<td>25 Hz</td>
<td>Butterworth</td>
</tr>
<tr>
<td>0x40 (64)</td>
<td>20 Hz</td>
<td>Butterworth</td>
</tr>
<tr>
<td>0x50 (80)</td>
<td>10 Hz</td>
<td>Butterworth</td>
</tr>
<tr>
<td>0x60 (96)</td>
<td>5 Hz (default)</td>
<td>Butterworth</td>
</tr>
</tbody>
</table>

12.6 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 defined 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, 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
  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”
} UserConfigurationStruct;

Default configuration

Default system parameters reside in the gDefaultUserConfig structure in file UserConfiguration.c. They are becoming active each time a 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)

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>MSB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>MSB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>MSB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
4. Mapping of 8-byte double value into 64-bit parameter (value is in Little Endian format)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>MSB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

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 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.

12.7 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.

### 12.7.1 Get algorithm input

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

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

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

### 12.7.2 Run the algorithm

A user defined algorithm should provide its main procedure as:

```c
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:

```c
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;
}
```
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

```c
// 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:

```c
#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.

```c
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.

### 12.7.3 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**.

```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.

```c
// 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\tax:%.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.

### 12.7.4 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.

```c
// 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
    USR_OUT_MAX     // place output packet definitions here
}UserOutPacketType;
```

- Add encoding procedure in `UserMessaging.c`.

User defined packets are encoded by this procedure:

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

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

```c
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)
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 = YYY; // 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;
}

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 `dataProcessingAndPresentation.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.
```

(continues on next 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

```
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.
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

### 13.1 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 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).

13.2 Leveler App

Leveler App Description - To Be Provided

13.3 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. 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.

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 and a heading 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 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 high-gain initialization phase. During the initialization phase, the OpenIMU unit is expected to be stationary or quasi-static so the EKF weights the accelerometer gravity reference and Earth’s magnetic field reference heavily in order to rapidly estimate the X, Y, and Z rate sensor bias, and the initial attitude and heading of the AHRSx81ZA. The initialization phase lasts approximately 60 seconds, and the initialization phase can be monitored in the softwareStatus BIT transmitted by default in each measurement packet. After the initialization phase, the AHRS/VP APP operates with lower levels of 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.

**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. See section XXX for information and tips regarding installation and calibration and why magnetic calibration is necessary. Please review this section of the manual before proceeding to using the heading data.

### 13.4 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 3-axis velocity information, as well as heading, roll, and pitch measurements, in addition to digital IMU data.

At a fixed N rate, the OpenIMU continuously maintains the digital IMU data; the dynamic roll, pitch, and heading data; as well as the navigation data. As shown in the software block the calibrated IMU data is passed into an “Integration to Orientation” block. The “Integration to Orientation” block integrates body frame sensed angular rate 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. Following the integration to orientation block, the body frame accelerometer signals are rotated into the NED level frame and are integrated to velocity. At this point, the data is blended with GPS position data, and output as a complete navigation solution.

As shown in the diagram, the Integration to Orientation and the Integration to Velocity signal processing blocks receive drift corrections from the Extended Kalman Filter (EKF) drift correction module. The drift correction module uses data from the aiding sensors, when they are available, to correct the errors in the velocity, attitude, and heading outputs. Additionally, when aiding sensors are available corrections to the rate gyro and accelerometers are performed.

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.

The status of GPS signal acquisition can be monitored from the hardwareStatus BIT in INS Apps Built in Test. From a cold start, it typically takes 40 seconds for GPS to lock. The actual lock time depends on the antenna’s view of the sky and the number of satellites in view.

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 and magnetometers to compute the initial roll, pitch and yaw angles. The roll and pitch attitude will be initialized using the accelerometer’s reference of gravity, and yaw will be initialized using the leveled magnetometers X and Y axis reference of the earth’s magnetic field. 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 softwareStatus BIT transmitted by default in each measurement packet. After the initialization phase, the INS APP operates with lower levels of feedback (also referred to as EKF gain) from the GPS, accelerometers, and magnetometers.

Note: For proper operation, the INS APP relies on magnetic field readings from its internal 3-axis magnetometer. The INS APP must be installed correctly and calibrated for hard-iron and soft iron effects to avoid any system performance degradation. See section 3.4.1 for information and tips regarding installation and calibration and why magnetic calibration is necessary. Please review this section of the manual before proceeding to use the INS APP.

Note: For optimal performance the INS APP utilizes GPS readings from an external GPS receiver. The GPS receiver requires proper antennae installation for operation. See section 2.1.4 for information and tips regarding antenna installation.

Note: The OpenIMU300Rx (RI and RA) do not provide inputs for GPS or other external sensors, so they can’t support the INS App.
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\(^1\). In a VG, AHRS, or INS\(^2\) 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

\(^1\) This discussion presupposes a certain amount of knowledge. Details related to differential equations, linear algebra, multi-variable calculus, stochastic processes, etc. are not described.

\(^2\) 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)\(^1\). The equations to come use the superscripts listed in Table 1 to specify the frame in which a variable is measured.

<table>
<thead>
<tr>
<th>Frame</th>
<th>Superscript</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECEF-Frame</td>
<td>E</td>
<td>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 (\textit{not shown in Figure 1})</td>
</tr>
<tr>
<td>NED-Frame</td>
<td>N</td>
<td>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 Figure 1.</td>
</tr>
<tr>
<td>Perp-Frame</td>
<td>(\perp)</td>
<td>Frame aligned with the local tangent-frame ((z_{\perp})-axis parallel to the gravity vector). Dark blue lines in Figure 1.</td>
</tr>
<tr>
<td>Body-Frame</td>
<td>B</td>
<td>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 Figure 1.</td>
</tr>
</tbody>
</table>

\(\text{Figure 1}\) 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\(^\circ\), a pitch of 10\(^\circ\), and a heading of 30\(^\circ\). 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.

\(^1\) 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.
14.2 Attitude Parameters

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
4. Mathematical Relationships between Attitude Parameters
5. Attitude Parameters Example

14.2.1 Direction Cosine Matrices

The first of these, the direction cosine matrix\(^1\), \(N^B\), specifies the relationship of one frame relative to another by relaying how the basis-vectors of one frame relate to the basis-vectors of another. These matrices have the property

\(^1\) 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\(^2\) and the formulation used by Thomas Kane\(^3\) are relied upon extensively.

### 14.2.2 Quaternion Elements

The second parameter used to convey orientation information are quaternion elements\(^4\) (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:

\[ N \vec{q}^B = [q_0 \ q_v]^T \]

\[ = [q_0 \ q_1 \ q_2 \ q_3]^T \]

\[ = [\cos (\frac{\theta}{2}) \ \hat{u} \cdot \sin (\frac{\theta}{2})]^T \]

### 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\(^5\), 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\(^6\). 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:

\[ N \vec{q}^B = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 \ 2 \cdot (q_1 \cdot q_2 - q_0 \cdot q_3) \ 2 \cdot (q_1 \cdot q_3 - q_0 \cdot q_2) \ 2 \cdot (q_2 \cdot q_3 + q_0 \cdot q_1) \end{bmatrix} \]

---

\(^2\) 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: \(N R^L = [\hat{z}_L^N \ \hat{y}_L^N \ \hat{z}_L^N]\).

\(^3\) 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.

\(^4\) 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, \(q_v = \hat{u} \cdot \sin (\theta/2)\), occupies elements 2, 3, and 4.

\(^5\) The 321-rotation sequence is the only rotation sequence considered in this paper.

\(^6\) 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.
$NR^B$ can also be expressed in terms of Euler-angles, $N\vec{\Theta}^B = [\perp \phi^B \perp \theta^B N\psi^\perp]^T$:

$$NR^B = \begin{bmatrix} \cos (N\psi^\perp) & -\sin (N\psi^\perp) & 0 \\ \sin (N\psi^\perp) \cos (N\psi^\perp) & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos (\perp \theta^B) & \sin (\perp \theta^B) \cdot \sin (\perp \phi^B) & \sin (\perp \theta^B) \cdot \cos (\perp \phi^B) \\ 0 & \cos (\perp \phi^B) & -\sin (\perp \phi^B) \\ -\sin (\perp \theta^B) \cdot \sin (\perp \phi^B) & \cos (\perp \theta^B) \cdot \sin (\perp \phi^B) & \cos (\perp \theta^B) \cdot \cos (\perp \phi^B) \end{bmatrix}$$

In this case, $NR^B$ 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):

$$NR^B = NR^\perp \cdot R^B$$

Finally, Euler angles, $N\vec{\Theta}^B$, can be expressed in terms of quaternion-elements, $N\vec{q}^B$:

$$\perp \phi^B = \text{atan2}(2 \cdot (q_2 \cdot q_3 + q_0 \cdot q_1), q_0^2 - q_1^2 - q_2^2 + q_3^2)$$

$$\perp \theta^B = -\text{asin}(2 \cdot (q_1 \cdot q_3 - q_0 \cdot q_2))$$

$$N\psi^\perp = \text{atan2}(2 \cdot (q_1 \cdot q_2 + q_0 \cdot q_3), q_0^2 + q_1^2 - q_2^2 - q_3^2)$$

Note: Due to the way the roll and pitch are separated from the heading, the Euler angles, $\perp \phi^B$, $\perp \theta^B$, and $N\psi^\perp$ 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:

$$ER^B = ER^N \cdot NR^\perp \cdot R^B$$

Each transformation describes how one coordinate frame is related to the next in the sequence of rotations.

1. $R^B$: Transformation from the (light-blue) body-frame to the (dark blue) local perpendicular-frame ($\perp$)
2. $NR^\perp$: Transformation from the (dark blue) $\perp$-frame to the (red) local NED-frame
3. $ER^N$: Transformation from the (red) NED-frame to the ECEF-frame (ECEF-Frame not shown; black line are latitude and longitude lines). $ER^N$ 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. $ER^N$, the transformation between the NED and Earth-frame (used in the INS formulation), is solely a function of ECEF location, $ER^N = f(r^E)$, and is based on the WGS84 model.
2. $NR^B$, 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), $NR^B = f(g, b)$.
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>
<thead>
<tr>
<th>Measurement</th>
<th>Sensor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>GPS</td>
<td>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.</td>
</tr>
<tr>
<td>Velocity</td>
<td>1) Accelerometer 2) GPS</td>
<td>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).</td>
</tr>
<tr>
<td>Roll/Pitch</td>
<td>1) Angular-Rate Sensor 2) Accelerometer</td>
<td>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).</td>
</tr>
<tr>
<td>Heading</td>
<td>1) Angular-Rate Sensor 2) Magnetometer 3) GPS</td>
<td>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.</td>
</tr>
</tbody>
</table>

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\(^1\), as well as sensor bias and noise. Simplified equations for the three sensors are provided below:

\[
\begin{align*}
\ddot{\omega}_{\text{meas}} &= \ddot{\omega}_{\text{true}} + \ddot{\omega}_{\text{bias}} + \ddot{\omega}_{\text{noise}} \\
\ddot{a}_{\text{meas}} &= \ddot{a}_{\text{motion}} + \ddot{a}_{\text{grav}} + \ddot{a}_{\text{bias}} + \ddot{a}_{\text{noise}} \\
\vec{m}_{\text{meas}} &= \vec{b}_{\text{motion}} + \vec{m}_{\text{bias}} + \vec{m}_{\text{noise}}
\end{align*}
\]

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:

\[
\ddot{\omega}_{\text{bias}} = \ddot{\omega}_{\text{offset}} + \ddot{\omega}_{\text{drift}}
\]

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”\(^2\) procedure once installed in the end-user’s system. The equations relating the hard and soft-iron effects\(^3\) on the measured magnetic field is:

\[
\vec{m}_{\text{meas}} = (R_{SI} \cdot S_{SI} \cdot R_{SI}^T)^{-1} \cdot \vec{b} + \vec{m}_{HI} + \vec{m}_{\text{bias}} + \vec{m}_{\text{noise}}
\]

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.

---

\(^1\) Due to the way the accelerometer measures acceleration, gravity appears like a deceleration and, as such, \(\ddot{a}_{\text{grav}} = -\ddot{g}\). 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.

\(^2\) 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).

\(^3\) 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

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 nonlinear 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. 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 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.

\[
\tilde{x}_{k|k-1} = f(\tilde{x}_{k-1|k-1}, \tilde{u}_{k|k-1})
\]

\[
P_{k|k-1} = F_{k-1} \cdot P_{k-1|k-1} \cdot F_{k-1}^T + Q_{k-1}
\]

The first equation (\(\tilde{x}_{k|k-1}\)) 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.

\[
\tilde{v}_k = \tilde{z}_k - \tilde{h}_k
\]
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.

\[
S_k = H_k \cdot P_{k|k-1} \cdot H_k^T + R_k \\
K_k = P_{k|k-1} \cdot H_k^T \cdot S_k^{-1} \\
\Delta \vec{x}_k = K_k \cdot \vec{v}_k \\
\vec{x}_{k|k} = \vec{x}_{k|k-1} + \Delta \vec{x}_k \\
\Delta P_k = -K_k \cdot H_k \cdot P_{k|k-1} \\
P_{k|k} = P_{k|k-1} + \Delta P_k
\]

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

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

\[
\vec{x} = \begin{bmatrix}
\vec{r}^N \\
\vec{v}^N \\
N \vec{q} \\
\vec{\omega}_B^{\text{bias}} \\
\vec{\alpha}_B^{\text{bias}}
\end{bmatrix} = \begin{bmatrix}
\text{NED Position (3)} \\
\text{NED Velocity (3)} \\
\text{Body Attitude (4)} \\
\text{Angular-Rate Bias (3)} \\
\text{Accelerometer Bias (3)}
\end{bmatrix}
\]

with the state-transition model, \( \vec{f} \), made up of five individual models (developed in upcoming sections):

\[
\vec{x}_k = \vec{f}(\vec{x}_{k-1}, \vec{u}_{k-1}) + \vec{w}_{k-1}
\]

1 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.
2 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:

\[
\vec{f}(\vec{x}_{k-1}, \vec{u}_{k-1}) = \begin{bmatrix}
\vec{v}_N^N + [N R^B_{k-1} \cdot \left( \vec{a}_{\text{meas, }k-1} - \vec{a}_{\text{bias, }k-1} \right) + \vec{a}_{\text{grav, }k-1}] \cdot dt \\
\Omega_{\text{meas, }k-1} - \Omega_{\text{bias, }k-1} \end{bmatrix} \cdot \vec{q}_{k-1}^B
\]

and the process-noise vector, \( \vec{w}_{k-1} \), is:

\[
\vec{w}_{k-1} = \begin{bmatrix}
-N R^B_{k-1} \cdot \vec{a}_{\text{noise}} \cdot dt^2 \\
-N R^B_{k-1} \cdot \vec{a}_{\text{noise}} \cdot dt \\
-N \cdot \vec{w}_{\text{noise}} \\
\vec{N}(0, \sigma^2_{dd,\omega}) \\
\vec{N}(0, \sigma^2_{dd,a})
\end{bmatrix}
\]

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 \( \sigma^2_{dd,\omega} \) or \( \sigma^2_{dd,a} \).

### 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:

\[
\vec{x}_k = \vec{x}_{k-1} + \frac{\vec{x}_{k-1} \cdot dt}{1!} + \frac{\vec{x}_{k-1} \cdot dt^2}{2!} + \ldots
\]

where terms higher than first-order are neglected. For attitude, the quaternion is propagated according to the expression:

\[
\vec{q}_k \approx \vec{q}_{k-1} + \frac{\vec{q}_{k-1} \cdot dt}{1!}
\]

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, \( \vec{q}_{k-1} \), is a function of true angular velocity, \( \vec{\omega}_{\text{true}} \), as follows:

\[
\vec{q}_{k-1} = \frac{1}{2} \cdot \Omega_{\text{true, }k-1} \cdot \vec{q}_{k-1}
\]

where \( \Omega_{\text{true, }k-1} \) is formed from the components of the angular rate vector, \( \left( N \vec{\omega}_{\text{true}}^B \right)^B \) 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 \( \vec{\omega}_{\text{true}} \).

\[
\vec{\omega}^B = \begin{bmatrix}
\omega^B_x \\
\omega^B_y \\
\omega^B_z
\end{bmatrix}
\]

The quaternion propagation matrix, \( \Omega_{k-1} \), at time-step k-1 is:

\[
\Omega_{k-1} = \begin{bmatrix}
0 & -\omega^B_y & -\omega^B_z & -\omega^B_{z,k-1} & -\omega^B_{y,k-1} & -\omega^B_{x,k-1} \\
\omega^B_z & 0 & -\omega^B_z & -\omega^B_{x,k-1} & -\omega^B_{z,k-1} & -\omega^B_{y,k-1} \\
\omega^B_y & \omega^B_{y,k-1} & 0 & -\omega^B_{y,k-1} & -\omega^B_{x,k-1} & -\omega^B_{z,k-1} \\
\omega^B_{z,k-1} & \omega^B_{z,k-1} & \omega^B_{y,k-1} & 0 & -\omega^B_{x,k-1} & -\omega^B_{z,k-1}
\end{bmatrix}
\]
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:

\[
\ddot{q}_k = \dot{q}_{k-1} + \frac{1}{2} \Omega_{true,k-1} \cdot \dot{q}_{k-1} \cdot dt = \left[ I_4 + \frac{dt}{2} \cdot \Omega_{true,k-1} \right] \cdot \dot{q}_{k-1}
\]

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:

\[
\ddot{q}_k = \left[ I_4 + \frac{dt}{2} \cdot (\Omega_{meas,k-1} - \Omega_{bias,k-1}) - \frac{dt}{2} \cdot \Omega_{noise,k-1} \right] \cdot \dot{q}_{k-1}
\]

\[
= \Phi_{k-1} \cdot \dot{q}_{k-1} + \vec{w}_{q,k-1}
\]

\( \Phi_{k-1} \) 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:

\[
\vec{w}_{q,k-1} = -\frac{dt}{2} \cdot \Omega_{noise,k-1} \cdot \dot{q}_{k-1}
\]

**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 (\( \omega^B_{noise} \)) forms the input vector, results in the final expression for the quaternion process-noise resulting from rate-sensor noise:

\[
\vec{w}_{q,k-1} = -\frac{dt}{2} \cdot \Xi_{k-1} \cdot \omega^B_{noise}
\]

with the variable \( \Xi_{k-1} \) relating the change in process noise to system attitude

\[
\Xi_{k-1} \equiv \begin{bmatrix}
q_0^T & -\vec{q}_v^T \\
-q_v \cdot [\vec{q}_v \times]
\end{bmatrix}
\]

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:

\[
\Sigma_{ij} = \text{cov} (\vec{x}_i, \vec{x}_j) = E \left[ (\vec{x}_i - \mu_i) \cdot (\vec{x}_j - \mu_j) \right]
\]

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 \( \Sigma_q \):

\[
\Sigma_q = \left( \frac{\sigma_\omega \cdot dt}{2} \right)^2 \cdot \begin{bmatrix}
1 - q_0^2 & -q_0 \cdot q_1 & -q_0 \cdot q_2 & -q_0 \cdot q_3 \\
-q_0 \cdot q_1 & 1 - q_1^2 & -q_1 \cdot q_2 & -q_1 \cdot q_3 \\
-q_0 \cdot q_2 & -q_1 \cdot q_2 & 1 - q_2^2 & -q_2 \cdot q_3 \\
-q_0 \cdot q_3 & -q_1 \cdot q_3 & -q_2 \cdot q_3 & 1 - q_3^2
\end{bmatrix}
\]

14.5. State Transition Models
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.

\[ \dot{\vec{a}}_{\text{motion},k-1} = \dot{\vec{a}}_{\text{meas},k-1} - \dot{\vec{a}}_{\text{bias},k-1} - \ddot{\vec{a}}_{\text{grav}} \]

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:

\[ \vec{v}^N_k = \vec{v}^N_{k-1} + (\dot{\vec{a}}^N_{\text{motion},k-1} - N R^B_{k-1} \cdot \dot{\vec{a}}^B_{\text{noise}}) \cdot dt \]

where

\[ \dot{\vec{a}}^N_{\text{motion},k-1} = N R^B_{k-1} \cdot (\dot{\vec{a}}^B_{\text{meas},k-1} - \dot{\vec{a}}^B_{\text{bias},k-1}) - \ddot{\vec{a}}^N_{\text{grav}} \]

The velocity process-noise vector resulting from accelerometer noise is:

\[ \vec{w}^N_{v,k-1} = -N R^B_{k-1} \cdot \dot{\vec{a}}^B_{\text{noise}} \cdot dt \]

leading to the final formulation for the velocity state-transition model:

\[ \vec{v}^N_k = \vec{v}^N_{k-1} + \dot{\vec{v}}^N_{k-1} \cdot dt + \vec{w}^N_{v,k-1} \]

The velocity process noise vector is used to compute the elements of the process covariance matrix (Q) related to the velocity estimate, as follows:

\[ \Sigma_v = \vec{w}^T_{v,k-1} \cdot \vec{w}_{v,k-1} \]

By making the assumption that all axes have the same noise characteristics \((\sigma^2_a)\) 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:

\[ \vec{r}_k = \vec{r}_{k-1} + \dot{\vec{r}}_{k-1} \cdot dt + \vec{w}_{r,k-1} \]

\( \vec{w}_{r,k-1} \) is the process noise associated with the position state-transition model, which is directly related to the velocity process noise:

\[ \vec{w}_{r,k-1} = \vec{w}_{v,k-1} \cdot dt \]

\[ = N R^B_{k-1} \cdot \dot{\vec{a}}^B_{\text{noise}} \cdot dt^2 \]

Like the previous process models, this expression is used to compute the elements of the process covariance matrix (Q) related to the position estimate:

\[ \Sigma_r = \vec{w}^T_{r,k-1} \cdot \vec{w}_{r,k-1} \]

By making the assumption that all axes have the same noise characteristics \((\sigma^2_a)\), \( \Sigma_r \) 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 ($\vec{\omega}_{offset}$)

2) A randomly varying component superimposed on the offset ($\vec{\omega}_{drift}$) based on the measured bias-instability value of the sensor

For the rate-sensor, the bias model is

$$\vec{\omega}_{bias} = \vec{\omega}_{offset} + \vec{\omega}_{drift}$$

The drift model follows a random-walk process\(^1\), i.e. the drift value wanders according to a Gaussian distribution.

$$\vec{\omega}_{drift,k} = \vec{\omega}_{drift,k-1} + \vec{\omega}_{drift, k-1} \cdot dt$$

where

$$\vec{\omega}_{drift, k-1} = N(0, \sigma_{dd, \omega})$$

Note: The subscript $dd$ stands for “drift-dot”.

Based on this model, the process variance for $\vec{\omega}_{drift}$ at time, $t$, is given by:

$$\sigma_{d, \omega}^2(t) = \left(\sigma_{dd, \omega} \cdot \sqrt{dt} \cdot \sqrt{t}\right)^2$$

An empirical study related $\sigma_{dd, \omega}$ to the BI and ARW values as follows:

$$\sigma_{dd, \omega} = 2 \cdot \frac{\pi}{\ln(2)} \cdot \frac{BI^2}{ARW}$$

To find the rate-bias process-noise covariance, set $t = dt$ in the process-variance model (above), resulting in:

$$\Sigma_{\omega_b} = \sigma_{d, \omega}^2(dt) \cdot I_3 = \left(\sigma_{dd, \omega} \cdot dt\right)^2 \cdot I_3$$

The accelerometer drift model mirrors this formulation and results in:

$$\Sigma_{\omega_a} = \sigma_{a, \omega}^2(dt) \cdot I_3 = \left(\sigma_{dd, a} \cdot dt\right)^2 \cdot I_3$$

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.

\(^1\) 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:

$$F_{k-1} = \left. \frac{\partial f^i}{\partial x} \right|_{\vec{x}_{k-1}, \vec{u}_{k-1}}$$

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:

$$F = I_{16} + \begin{bmatrix} 0_3 & I_3 & 0_{3 \times 4} & 0_3 & 0_3 & -N R^B \\ 0_3 & 0_3 & \frac{1}{2} \cdot \Omega & -\frac{1}{2} \cdot \Xi & 0_{4 \times 3} \\ 0_{4 \times 3} & 0_{4 \times 3} & 0_{3 \times 4} & 0_3 & 0_3 \\ 0_3 & 0_3 & 0_{3 \times 4} & 0_3 & 0_3 \\ 0_3 & 0_3 & 0_{3 \times 4} & 0_3 & 0_3 \end{bmatrix} \cdot dt$$

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:

$$\overline{Q}_F = \begin{bmatrix} q_1 & q_0 & -q_3 & q_2 \\ q_2 & q_3 & q_0 & -q_1 \\ q_3 & -q_2 & q_1 & q_0 \\ \vec{q}_v & q_0 \cdot I_3 + [\vec{q}_v \times] \end{bmatrix}$$

and

$$\vec{a}^B = \vec{a}^B_{meas} - \vec{a}^B_{bias}$$

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, $\vec{\omega}_k$ (found in previous sections), the Process Noise Covariance Matrix is:

$$Q_k = \begin{bmatrix} \Sigma_r & 0_3 & 0_{3 \times 4} & 0_3 & 0_3 \\ 0_3 & \Sigma_v & 0_{3 \times 4} & 0_3 & 0_3 \\ 0_{4 \times 3} & 0_{4 \times 3} & \Sigma_q & 0_{4 \times 3} & 0_{4 \times 3} \\ 0_3 & 0_3 & 0_{3 \times 4} & \Sigma_{\omega b} & 0_3 \\ 0_3 & 0_3 & 0_{3 \times 4} & 0_3 & \Sigma_{\omega b} \end{bmatrix}$$
The individual process covariance are repeated here:

$$\Sigma_r = (\sigma_a \cdot dt^2)^2 \cdot I_3$$
$$\Sigma_v = (\sigma_a \cdot dt)^2 \cdot I_3$$
$$\Sigma_q = \left( \frac{\sigma_{ω \cdot dB}}{2} \right)^2$$
$$\Sigma_w = (\sigma_{dd,ω} \cdot dt)^2 \cdot I_3$$
$$\Sigma_{ab} = (\sigma_{dd, a} \cdot dt)^2 \cdot I_3$$

## 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:

1. Accelerometers “level” the system (used to compute $\perp ϕ_{meas}$ and $\perp θ_{meas}$) FN
2. Magnetometers and/or GPS heading information align the $\perp$-frame with true or magnetic north ($N \perp ϕ$)
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:

$$\vec{z}_k = \begin{bmatrix} \vec{r}_{GPS}^N \\ \vec{v}_{GPS}^N \\ N乙方_{meas} \end{bmatrix}$$

with the corresponding measurement model, $\vec{h}_k$:

$$\vec{h}_k = \begin{bmatrix} \vec{r}_{pred}^N \\ \vec{v}_{pred}^N \\ N乙方_{pred} \end{bmatrix}$$

Both $N乙方_{meas}$ and $N乙方_{pred}$ 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 $N乙方_{pred}$ is computed from $N乙方_{pred}$, as follows:

$$\perp ϕ_{pred} = atan2 \left[ 2 \cdot (q_2 \cdot q_3 + q_0 \cdot q_1) , q_0^2 - q_1^2 - q_2^2 + q_3^2 \right]$$

$$\perp θ_{pred} = -asin \left[ 2 \cdot (q_1 \cdot q_3 - q_0 \cdot q_2) \right]$$

$$N乙方_{pred} = atan2 \left[ 2 \cdot (q_1 \cdot q_2 + q_0 \cdot q_3) , q_0^2 + q_1^2 - q_2^2 - q_3^2 \right]$$
Measurement Vector ($\mathbf{vec{z}}_k$)

The measurement vector, $\mathbf{z}_k$, 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 $-1 \text{ [g]}$. 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 $1 \text{ [g]}$.

$$\mathbf{a}_{meas} = \mathbf{a}_{grav} = -\mathbf{g}$$

Static roll and pitch values are determined by noting that gravity is constant in the N-Frame (perp-Frame):

$$\mathbf{g}^N = \mathbf{g}^\perp = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

and can be transformed into the body frame through $B^R \perp$:

$$\mathbf{g}^B = B^R \perp \cdot \mathbf{g}^\perp = (\perp R^B)^T \cdot \mathbf{g}^\perp = (\perp R^B)^T \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

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:

$$\mathbf{g}^B = -\mathbf{a}_{meas}$$

$$\begin{bmatrix} -\sin(\perp \theta^B) \\ \cos(\perp \theta^B) \cdot \sin(\perp \phi^B) \\ \cos(\perp \theta^B) \cdot \cos(\perp \phi^B) \end{bmatrix} = \begin{bmatrix} -a_{mx}^B \\ -a_{my}^B \\ -a_{mz}^B \end{bmatrix}$$

From this result, the angles corresponding to the accelerometer signal are found:

$$\perp \phi^B_{meas} = \arctan2(-a_{my}^B, -a_{mz}^B)$$
$$\perp \theta^B_{meas} = -\arcsin(-\hat{a}_{mx}^B)$$

where, $\hat{a}_{mx}^B$ is the x-axis acceleration value normalized by the total acceleration magnitude:

$$\hat{a}_{mx}^B = \frac{a_{mx}^B}{\|\mathbf{a}_{meas}^B\|}$$

Normalization of the y and z-axis accelerometer values can be performed. However this is not required as the $\arctan$ 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

14.7. Measurement Model
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:

\[
\vec{m}_{\text{corr}}^B = R_{SI} \cdot S_{SI} \cdot R_{SI}^T \cdot (\vec{m}_{\text{meas}}^B - \vec{m}_{\text{bias}}^B - \vec{m}_{HI}^B)
\]

where \(\vec{m}_{\text{meas}}^B\) 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 Components based on WMM](image)

**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
the NED-frame:

\[ \vec{b}_c^\perp = R_N^N \cdot \vec{b}_N^N = R_N^N \cdot \begin{\bmatrix} b_{xy} \\ 0 \\ b_z \end{bmatrix} \]

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 \( R_{pred}^B \).

\[ \vec{m}_{corr}^\perp = R_{pred}^B \cdot \vec{m}_{corr}^B \]

**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**

---

Fig. 3: **Relationship of Magnetic-Field to N and B-Frames**

---

14.7. Measurement Model 172
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 3.

Table 3: GPS Messaging and Heading Measurement

<table>
<thead>
<tr>
<th>System</th>
<th>Message</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>NovAtel</td>
<td>BESTVEL</td>
<td>Actual direction of motion over ground (track over ground) with respect to True North</td>
<td>[deg]</td>
</tr>
<tr>
<td></td>
<td>VTG</td>
<td>True track made good</td>
<td>[deg]</td>
</tr>
<tr>
<td>SiRF</td>
<td>Geodetic Navigation Data – Message ID 41</td>
<td>Course Over Ground (COG, True)</td>
<td>[deg x 100]</td>
</tr>
<tr>
<td>ublox</td>
<td>NAV-VELNED</td>
<td>Heading of motion 2-D</td>
<td>[deg]</td>
</tr>
</tbody>
</table>

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 $\perp_\theta^{B}_{meas}$ and $\perp_\theta^{B}$) FN

2) Magnetometers and/or GPS heading information align the perp-frame with true or magnetic north ($N_\theta^{\perp}$) FN

3) GPS position and velocity measurements update the position and velocity estimates ($\vec{r}^N$ and $\vec{v}^N$) FN

Based upon these steps, the measurement vector, $\vec{z}_k$, is formed:

$$\vec{z}_k = \begin{bmatrix} \vec{r}^N_{GPS} \\ \vec{v}^N_{GPS} \\ N_\theta^B_{meas} \end{bmatrix}$$

with the corresponding measurement model, $\vec{h}_k$:

$$\vec{h}_k = \begin{bmatrix} \vec{r}^N_{pred} \\ \vec{v}^N_{pred} \\ N_\theta^B_{pred} \end{bmatrix}$$

Both $N_\theta^B_{meas}$ and $N_\theta^B_{pred}$ 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 $B R^{\perp}$:

$$\vec{g}^B = B R^{\perp} \cdot \vec{g}^{\perp} = \left(\perp R^B\right)^T \cdot \vec{g}^{\perp} = \left(\perp R^B\right)^T \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

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:

$$\vec{g}^B = -\vec{a}^B_{meas}$$
\[
\begin{align*}
\begin{bmatrix}
-\sin(\theta^B) \\
\cos(\theta^B) \cdot \sin(\phi^B) \\
\cos(\theta^B) \cdot \cos(\phi^B)
\end{bmatrix}
&= \begin{bmatrix}
-a^B_{mx} \\
a^B_{my} \\
a^B_{mz}
\end{bmatrix}
\end{align*}
\]

From this result, the angles corresponding to the accelerometer signal are found:

\[
\begin{align*}
\hat{\phi}^B_{\text{meas}} &= \arctan2(-a^B_{my}, -a^B_{mz}) \\
\hat{\theta}^B_{\text{meas}} &= -\sin(-\hat{a}^B_{mx})
\end{align*}
\]

where, \(\hat{a}^B_{mx}\) is the x-axis acceleration value normalized by the total acceleration magnitude:

\[
\hat{a}^B_{mx} = \frac{a^B_{mx}}{\|\hat{a}^B_{\text{meas}}\|}
\]

Normalization of the y and z-axis accelerometer values can be performed. However this is not required as the \(\arctan\) 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:

\[
\vec{m}^B_{\text{corr}} = R_{SI} \cdot S_{SI} \cdot R_{SI}^T \cdot (\vec{m}^B_{\text{meas}} - \vec{m}^B_{\text{bias}} - \vec{m}^B_{HI})
\]

where \(\vec{m}^B_{\text{meas}}\) is the measured magnetic field vector in the body-frame, \(\vec{m}^B_{HI}\) 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:
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:

\[
\vec{b}^\perp = \pm R^N \cdot \vec{b}^N = \pm R^N \cdot \begin{pmatrix} b_{xy} \\ 0 \\ b_z \end{pmatrix}
\]

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:

\[
\tan(N \psi^\perp) = \frac{b_{xy} \cdot \sin(N \psi^\perp)}{b_{xy} \cdot \cos(N \psi^\perp)} = \frac{-b_y^\perp}{b_x^\perp} = \frac{-m_{corr,y}^B}{m_{corr,x}^B}
\]

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 \( \perp R^B_{pred} \).

\[
\vec{m}_{corr}^\perp = \perp R^B_{pred} \cdot \vec{m}_{corr}^B
\]

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>
<thead>
<tr>
<th>System</th>
<th>Message</th>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>NovAtel</td>
<td>BESTVEL</td>
<td>Actual direction of motion over ground (track over ground) with respect to True North</td>
<td>[deg]</td>
</tr>
<tr>
<td>NMEA</td>
<td>VTG</td>
<td>True track made good</td>
<td>[deg]</td>
</tr>
<tr>
<td>SiRF</td>
<td>Geodetic Navigation Data – Message ID 41</td>
<td>Course Over Ground (COG, True)</td>
<td>[deg x 100]</td>
</tr>
<tr>
<td>ublox</td>
<td>NAV-VELNED</td>
<td>Heading of motion 2-D</td>
<td>[deg]</td>
</tr>
</tbody>
</table>

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:

\[ v_N = v_{XY} \times \cos(\psi^N) \]
\[ v_E = v_{XY} \times \sin(\psi^N) \]

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 measurement 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), \( \tilde{y}_k \), is computed:

\[ \tilde{y}_k = \tilde{z}_k - \tilde{h}_k \]

This result is used in the update stage of the EKF to generate the state error, \( \Delta \tilde{x}_k \), given the Kalman gain matrix.

The available sensor information is used as follows:

1. Accelerometers “level” the system (used to compute \( \phi^B_{meas} \) and \( \theta^B_{meas} \))
2. Magnetometers and/or GPS heading information align the perp-frame with true or magnetic north (\( \psi^N \))
3. GPS position and velocity measurements update the position and velocity estimates (\( \tilde{r}^N \) and \( \tilde{v}^N \))

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**

<table>
<thead>
<tr>
<th>Ellipse Parameter</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center</td>
<td>-0.128, 0.126</td>
<td>[G]</td>
</tr>
<tr>
<td>Major/Minor axes</td>
<td>0.225, 0.198</td>
<td>[G]</td>
</tr>
<tr>
<td>Soft-Iron Scale Factor</td>
<td>0.882</td>
<td>[N/A]</td>
</tr>
<tr>
<td>Angle to Major-Axis</td>
<td>-48.497</td>
<td>[deg]</td>
</tr>
</tbody>
</table>

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:

$$R_{SI} = \begin{bmatrix} \cos(\eta) & -\sin(\eta) & 0 \\ \sin(\eta) & \cos(\eta) & 0 \end{bmatrix}$$

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:

$$S_{SI} = \begin{bmatrix} 1/a & 01/b & 001 \end{bmatrix}$$

$a$ 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:

$$R_{SI} = \begin{bmatrix} 0.66266 & -0.748920 \end{bmatrix} 0.748920.662660 001$$

and

$$S_{SI} = \begin{bmatrix} 4.452260 \end{bmatrix} 05.046890 001$$
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**

<table>
<thead>
<tr>
<th>True Heading [deg]</th>
<th>Disturbance-Free Data</th>
<th>Data with Added Iron Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>359.69</td>
<td>-0.31</td>
</tr>
<tr>
<td>45</td>
<td>45.19</td>
<td>0.19</td>
</tr>
<tr>
<td>90</td>
<td>89.96</td>
<td>-0.04</td>
</tr>
<tr>
<td>135</td>
<td>135.05</td>
<td>0.05</td>
</tr>
<tr>
<td>180</td>
<td>180.57</td>
<td>0.57</td>
</tr>
<tr>
<td>225</td>
<td>225.64</td>
<td>0.64</td>
</tr>
<tr>
<td>270</td>
<td>270.63</td>
<td>0.63</td>
</tr>
<tr>
<td>315</td>
<td>315.30</td>
<td>0.30</td>
</tr>
<tr>
<td>360</td>
<td>359.79</td>
<td>-0.21</td>
</tr>
</tbody>
</table>

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-fixture 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**


### 14.11.2 Extended Kalman Filter Implementations


### 14.11.3 Mathematical References

**Process Models**


**Measurement Models**


Innovation (Measurement Error)


Magnetic-Alignment


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 VI

Miscellaneous
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 generated required data for the algorithms, run the algorithms, plot simulation results, save simulations results, and generate a brief summary.

GitHub Link: GNSS-INS-SIM

Use the browser’s back button to return.
CHAPTER 17

Python Serial Driver

Aceinna OpenIMU python driver, data logging, and web socket server
GitHub Link: python-openimu

Note: Use the browser’s back button to return to the OpenIMU documentation
CHAPTER 18

C-Code Serial Driver

C-code Serial Driver - Details To Be Provided
Index

E

environment variable
   ACCELERATION, 42, 50, 56
   ANGULARRATE, 42, 50, 56
   ELECTRICAL, 43, 57
   ENVIRONMENT, 43, 57
   MAGNETICFIELD, 43, 50
   PHYSICAL, 43, 57