openINS

Interface Control Document

Subject: *This document details the operation and communication of the openINS software package. The document uses a working example of openINS for the operator to build the program and use its functions in real time.*

Revision History

Revision	*Initials*	*Date*	*Comments*
001	Ljd	12/19/13	Initial.
002	Bwk	01/01/14	Initial review.
003	Ljd	01/02/14	Added content.
004	Bwk	01/09/14	Edited.
005	Ljd	01/09/14	Edited.
006	Meb	08/26/14	Edited.
007	Jdr	09/23/14	Added content.
008	Clr	09/24/14	Edited.
009	Clr	09/24/14	Edited.
010	Clr	06/05/15	Added content and edited.
011	Meb	06/17/15	Proofread and edited.

Table 1: Revision history.

Table of Contents

Index of Tables

Illustration Index

openINS

Interface Control Document

1 Introduction

openINS provides a complete software implementation of an aided Inertial Navigation System (INS) suited primarily for unmanned underwater vehicles. The software package runs on Greensea's proprietary technology, openSEA, and is part of the foundation our Balefire control and navigation system is built upon.

openINS is the core computational engine, the true muscle, behind both commercially packaged and vehicle-integrated navigation systems. It offers a robust and scalable computational platform that utilizes the device library in openSEA to support thousands of different devices, many of which are navigation sensors.

1.1 What We're Doing

This document goes in depth on the openINS software application. A working example will guide you through the set up and use of the system.

1.2 More about openINS

We at Greensea developed, use, and own openSEA. openINS is born from the openSEA core library. The end game? A flexible and powerful development technology for unmanned underwater vehicles providing device interfaces, navigation, control, and a robust software architecture.

openINS provides a robust multi-state Kalman filter that fuses inertial measurements from the IMU with data from aiding sensors to create an accurate final solution of the vehicle's state.

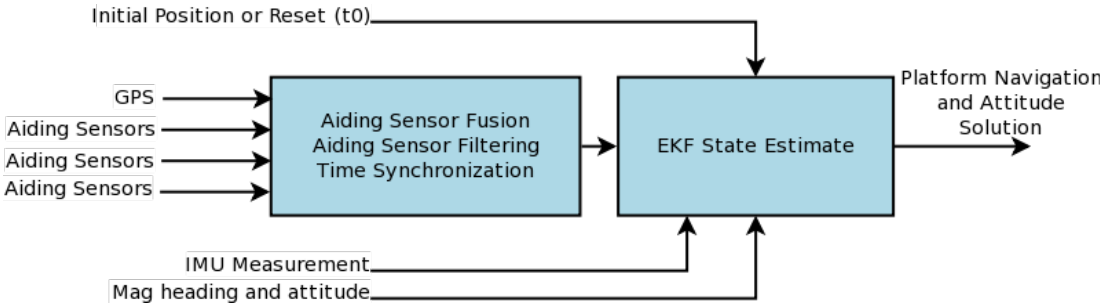

Illustration 1: Inertial Navigation System diagram.

2 A Note on Navigation

Navigation for underwater vehicles is a nightmare because, unlike with terrestrial or air applications, there's no way to accurately measure the absolute position of the system in the Earth Centered Earth Fixed (ECEF) coordinate frame. Above water, GPS gives an accurate "absolute" measurement of a system's position within the ECEF frame in units of latitude and longitude. Acoustic ranging techniques like USBL and LBL provide absolute position measurements with sometimes reasonable accuracies based on acoustic ranging and bearing between beacons. But these techniques often have large noise components and are heavily dependent on environmental and operational factors. Subsea, the best we can do is to take observations of as many states of the vehicle from as many different sources as possible and combine them to produce an *estimate* of the vehicle's position based on a known starting point.

It's like playing a very expensive game of Marco Polo, where you're blindfolded and forced to find your way based only on what directions people shout at you. Who ever thought that game was fun? It's not.

2.1 Sensor Fusion

As mentioned, we can observe many discrete states of an underwater vehicle by using inputs from several sensors on the vehicle. That might sound swell, but each sensor and its associated measurements contain noise, error, drift, and latency, so no single instrument can produce a navigation estimate by itself. The obvious answer seems to lie in combining the measurements, but beware of obvious answers — in attempting sensor fusion in an INS we have to manage noise, understand drift and bias, and account for measurement latency and differences in measurement rates. The true pain is, each sensor is in its own coordinate frame and the sensor frame has to be tied back to the vehicle frame, and then the ECEF frame, to provide positional information that's actually meaningful.

Sensor	*Useful DOF Data*	*Limits*	*Units*	*Orientation*
GPS	X,Y,Z	Surface only Low update rate	lat,lon	T^{B}_{gps}
USBL	X,Y,Z	Large errors Low update rate	lat,lon	T^{B}_{usbl}
Depth	Z	High-frequency noise	m	T^{B}_{dpt}
Altimeter	Zref	High-frequency noise	m	T^{B}_{dpt}
AHRS Compass	Φ, θ, Ψ $\overline{\Phi}, \overline{\theta}, \overline{\Psi}$	Magnetometer-based	rad rad/sec	R^{b}_{AHRS}
IMU	$\overline{\overline{x}}, \overline{\overline{y}}, \overline{\overline{z}}$ $\overline{\phi}, \overline{\Theta}, \overline{\psi}$	High drift Entirely dependent on initial conditions	m/sec^2 rad/sec	R^{b}_{imu}, T^{B}_{imu}
DVL	$x_{dot}, y_{dot}, z_{dot}$	Requires bottom lock Requires stable platform	m/s m/s m	T^{B}_{dvl}

Table 2: Sensors used in subsea navigation and their contribution to the final navigational estimate.

2.2 Coordinate Frames

There are a whole bunch of coordinate frames considered in developing a coherent navigation solution. Sensors such as compass, IMU, and DVL make orientation-relative measurements so their output must be transformed to the vehicle reference frame in order to incorporate them into the navigation solution. The vehicle reference frame also needs to be converted to a global reference frame in order to be meaningful. ECEF is that chosen frame.

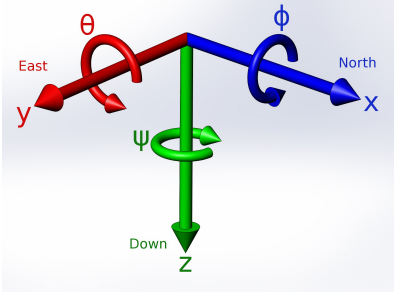

Illustration 2: NED coordinate frame.

For ECEF coordinates, a north-east-down (NED) coordinate system is used where north is defined as positive X and east is defined as positive Y. Increasing Z is greater depth, following the right hand rule. For the vehicle coordinate system, positive X is looking forward, positive Y is out the port side and positive Z is again downward. Rotations about these three axes are defined as Φ, θ, and Ψ respectively. You can see the NED coordinate system in the lovely diagram provided, and the vehicle coordinate system matches if you just replace north with “forward” and east with “port.”

3 Communicating with openINS

An INS collects data from aiding sensors such as Doppler Velocity Logs (DVLs), Ultra-Short Baseline (USBL) tracking systems, altimeters, and GPS units to provide a precise and high-rate position for subsea vehicles. The sensor signals are received as an input to the system. The system considers IMU, attitude, and depth data together with the aiding sensor data, and feeds it to a state estimation utility where the navigation state is calculated. The output is the navigation solution.

3.1 LCM Communication Channels

The openINS application will publish data messages continuously on channels while it's running. The input channels send data to the system and the output channels send data out. To interact with openINS you can either publish data on a channel it subscribes to or subscribe to one of the channels it publishes.

Voila:

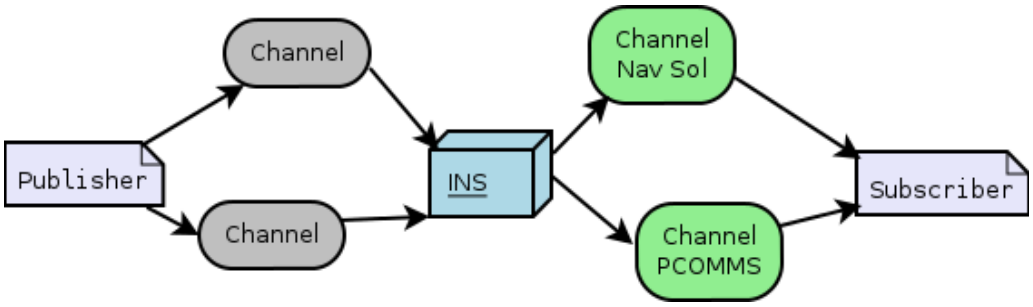

Illustration 3: openINS communication with LCM channels and message passing.

The data sent through these channels has been encoded using structures called LCM types.

3.2 PCOMMS Commands

openINS has a PCOMMS-based command interface for setting navigation parameters and system configurations. The PCOMMS command interface uses the pcomms_t LCM type. openINS subscribes to a PCOMMS command channel and listens for incoming PCOMMS commands. The commands are arbitrated by the name field in the three native pcomms_t types: analog_t, digital_t, and message_t. Below is a table outlining the command names:

Command Name	*Type*	*Description*
U_MAG_DEC	analog_t	Set the magnetic declination (degrees).
U_CAL_DEPTH	digital_t	Begin a depth calibration. To start the calibration, send one digital_t message with U_CAL_DEPTH set to “true”. Calibration will automatically end.
U_INIT_POS	digital_t	Set the position of the system to the values of U_INIT_LAT and U_INIT_LON. The pcomms_t message must include the three U_INIT variables: U_INIT_POS, U_INIT_LAT, and U_INIT_LON.
U_INIT_LAT	analog_t	The latitude to set the systems position (decimal degrees).
U_INIT_LON	analog_t	The longitude to set the systems position (decimal degrees).
U_IMU_BIAS_X	analog_t	Set the IMU X bias.

Command Name	*Type*	*Description*
U_IMU_BIAS_Y	analog_t	Set the IMU Y bias.
U_IMU_BIAS_Z	analog_t	Set the IMU Z bias.
U_IMU_BIAS_PHI	analog_t	Set the IMU phi bias.
U_IMU_BIAS_THETA	analog_t	Set the IMU theta bias.
U_IMU_BIAS_PSI	analog_t	Set the IMU psi bias.
U_CMPS_BIAS_X	analog_t	Set the compass X bias.
U_CMPS_BIAS_Y	analog_t	Set the compass Y bias.
U_CMPS_BIAS_Z	analog_t	Set the compass Z bias.
U_CMPS_BIAS_PHI	analog_t	Set the compass phi bias.
U_CMPS_BIAS_THETA	analog_t	Set the compass theta bias.
U_CMPS_BIAS_PSI	analog_t	Set the compass psi bias.
U_DVL_BIAS_X	analog_t	Set the DVL X bias.
U_DVL_BIAS_Y	analog_t	Set the DVL Y bias.
U_DVL_BIAS_Z	analog_t	Set the DVL Z bias.
U_DVL_BIAS_PHI	analog_t	Set the DVL phi bias.
U_DVL_BIAS_THETA	analog_t	Set the DVL theta bias.
U_DVL_BIAS_PSI	analog_t	Set the DVL psi bias.
U_DEPTH_BIAS_X	analog_t	Set the depth X bias.
U_DEPTH_BIAS_Y	analog_t	Set the depth Y bias.
U_DEPTH_BIAS_Z	analog_t	Set the depth Z bias.
U_DEPTH_BIAS_PHI	analog_t	Set the depth phi bias.
U_DEPTH_BIAS_THETA	analog_t	Set the depth theta bias.
U_DEPTH_BIAS_PSI	analog_t	Set the depth psi bias.
U_GPS_BIAS_X	analog_t	Set the GPS X bias.
U_GPS_BIAS_Y	analog_t	Set the GPS Y bias.
U_GPS_BIAS_Z	analog_t	Set the GPS Z bias.
U_GPS_BIAS_PHI	analog_t	Set the GPS phi bias.
U_GPS_BIAS_THETA	analog_t	Set the GPS theta bias.
U_GPS_BIAS_PSI	analog_t	Set the GPS psi bias.
U_CHANNEL_IMU	message_t	Set the IMU receive channel to the message_t value.
U_CHANNEL_COMPASS	message_t	Set the compass receive channel to the message_t value.
U_CHANNEL_PRESSURE	message_t	Set the pressure receive channel to the message_t value.
U_CHANNEL_GUI_PCOMMS	message_t	Set the GUI PCOMMS receive channel to the message_t value.
U_CHANNEL_MARK	message_t	Set the mark receive channel to the message_t value.

Command Name	Type	Description
U_CHANNEL_DVL	message_t	Set the DVL receive channel to the message_t value.
U_CHANNEL_ALTIMETER	message_t	Set the altimeter receive channel to the message_t value.
U_CHANNEL_USBL	message_t	Set the USBL receive channel to the message_t value.
U_CHANNEL_GPS	message_t	Set the GPS receive channel to the message_t value.
U_CHANNEL_DEPTH	message_t	Set the depth receive channel to the message_t value.

Table 3: PCOMMS commands.

3.3 Important LCM Types

So, LCM types are data types specific to the system configuration. Each type houses a collection of definitions for the relevant values, states, and commands.

In the next sections, we've outlined the LCM types you'd need for openINS.

3.3.1 altitude_stat_t

altitude_stat_t is the LCM type used to calculate system altitude.

Variable	Type	Description
time_unix_sec	double	Unix Time stamp. The message data time stamp in seconds since January 1, 1970, 12 a.m. GMT.
count_publish	int64_t	Incremented count value for the number of times the application published this message.
count_publish_error	int64_t	Error count value for the number of times an error was encountered during message transmissions. This includes “No communications” and device errors.
id_device	string	The device identification string. This ID is the manufacturer and model number for a hardware device.
num_analogs	int32_t	Number of analog values.
analogs[num_analogs]	analog_t -vector	Vector of analog values. The vector contains a name and a double value.
altitude_valid	boolean	Altitude valid when Boolean is “true.”
altitude_m	double	The altitude value.

Table 4: LCM type altitude_stat_t.

3.3.2 compass_stat_t

The compass_stat_t LCM type is used to provide the data for attitude calculation.

Variable	Type	Description
time_unix_sec	double-	Unix Time stamp. The message data time stamp in seconds since January 1 1970, 12 a.m. GMT.

Variable	*Type*	*Description*
count_publish	int64_t	Incremented count value for the number of times the application published this message.
count_publish_error	int64_t	Error count value for the number of times an error was encountered during message transmissions. This includes “No communications” and device errors.
num_analogs	int32_t	The number of analog values.
analog_t[num_analogs]	analog_t - vector	Vector of analog values. The vector contains a name and a double value. This is used for device specific information and is not essential to the function of openINS.
id_device	string-	The device identification string. This ID is the manufacturer and model number for a hardware device.
attitude_valid	boolean	The rotational degree of freedom. “False” if attitudes are not valid.
attitude_roll__deg	double	The degree of roll.
attitude_pitch_deg	double	The degree of pitch.
attitude_heading_deg	double	The degree of current heading.

Table 5: LCM type compass_stat_t.

3.3.3 dvl_stat_t

The dvl_stat_t LCM type inputs the Doppler Velocity Log data to the system.

Variable	*Type*	*Description*
unix_time_sec	double	Unix Time stamp. The message data time stamp in seconds since January 1, 1970, 12 a.m. GMT.
count_publish	int64_t	Incremented count value for the number of times the application published this message.
count_publish_error	int64_t	Error count value for the number of times an error was encountered during message transmissions. This includes “No communications” and device errors.
num_analogs	int32_t	The number of analog values.
analogs[num_analogs]	analog_t	Vector of analog values. The vector contains a name and a double value. This is used for device specific information and is not essential to the function of openINS.
id_device	string	The device identification string. This ID is the manufacturer and model number for a hardware device.
lock_btm	boolean	Bottom lock. If not “true”, velocities and altitude are not valid.
velocity_btm_x_m_sec velocity_btm_y_m_sec velocity_btm_z_m_sec	double	Linear bottom velocity values in the X,Y, and Z directions.

Variable	*Type*	*Description*
altitude_btm_m altitude_btm_beam1_m altitude_btm_beam2_m altitude_btm_beam3_m altitude_btm_beam4_m	double	Bottom altitude values, mean and 4 beam values. (No bottom lock returns 999.9.)
velocity_ref_x_m_sec velocity_ref_y_m_sec velocity_ref_z_m_sec	double	Linear reference velocity value in the X, Y, and Z directions.
altitude_ref_m altitude_ref_beam1_m altitude_ref_beam2_m altitude_ref_beam3_m altitude_ref_beam4_m	double	Reference altitude values, mean value, and values from beams 1-4. (No bottom lock returns 999.9)
lock_ref	boolean	The DVL reference lock will only be valid if lock is “true”.

Table 6: LCM type dvl_stat_t.

3.3.4 imu_stat_t

The imu_stat_t LCM type describes the inertial measurement unit data set for openINS.

Variable	*Type*	*Description*
time_unix_sec	double	Unix Time stamp. The message data time stamp in seconds since January 1, 1970, 12 a.m. GMT.
count_publish	int64_t	Incremented count value for the number of times the application published this message.
count_publish_error	int64_t	Error count value for the number of times an error was encountered during message transmissions. This includes “No communications” and device errors.
id_device	string	The device identification string. This ID is the manufacturer and model number for a hardware device.
num_analogs	int32_t	Number of analog signals.
analogs[num_analogs]	analog_t – vector	Vector of analog values. The vector contains a name and a double value.
attitude_valid	boolean	Rotational degree of freedom attitude validity check.
attitude_roll_deg	double	The degree of roll.
attitude_pitch_deg	double	The degree of pitch.
attitude_heading_deg	double	The degree of heading.
rotational_vel_roll_deg_sec	double	Rotational degree of freedom in the IMU reference frame roll variable.
rotational_vel_pitch_deg_sec	double	Rotational degree of freedom in the IMU reference frame pitch variable.

Variable	*Type*	*Description*
rotational_vel_heading_deg_sec	double	Rotational degree of freedom in the IMU reference frame heading variable.
rotational_vel_valid	boolean	The rotational velocity reading is valid if "true".
acceleration_valid	boolean	The acceleration (meters per second) reading is valid if "true".
acceleration_x_m_sec2	double	The acceleration (meters per second) in the IMU reference frame variable for the X direction.
acceleration_y_m_sec2	double	The acceleration (meters per second) in the IMU reference frame variable for the Y direction.
acceleration_z_m_sec2	double	The acceleration (meters per second) in the IMU reference frame variable for the Z direction.
status	int32_t	Bitpacked integer that is IMU specific, not used by openINS.

Table 7: LCM type imu_stat_t.

3.3.5 nav_solution_t

The nav_solution_t LCM type is created by the data collected by openINS. The state estimate of the system is transmitted via this type.

Variable	*Type*	*Description*
unix_time	double	Unix Time stamp. The message data time stamp in seconds since January 1, 1970, 12 a.m. GMT.
attitude[3]	double	Attitude degrees of phi, theta, and psi, respectively.
attitude_dot[3]	double	Attitude rate of change.
count_publish	int64_t	Incremented count value for the number of times the application published this message.
vehicle	string	Vehicle ID.
relative_position[3]	double	Relative position in meters: north, east, down in 0,1,2, respectively.
relative_position_dot[3]	double	Relative position rate of change in meters per second: north, east, down in 0,1,2, respectively.
relative_acceleration[3]	double	Relative acceleration.
absolute_position[3]	double	Absolute Position in decimal degrees longitude, latitude, and depth (WGS84 datum) in 0,1,2, respectively.
speed_over_ground	double	Speed over ground in meters per second.
course_over_ground	double	Course over ground.
altitude_above_bottom	double	Altitude above bottom in meters.
depth	double	Depth in meters.
initial_lonlat_fix[2]	double	Initial fix of longitude and latitude in decimal degrees (WGS84 datum).

Variable	*Type*	*Description*
last_lonlat_fix[2]	double	Last fix in decimal degrees.
imu_error	double	Placeholder for estimated IMU error.
dvl_error	double	Placeholder for estimated DVL error.
attitude_ok	boolean	Navigation system status check for attitude.
depth_ok	boolean	Navigation system status check for depth.
speed_ok	boolean	Navigation system status check for speed.
altitude_ok	boolean	Navigation system status check for altitude.
relative_position_ok	boolean	Navigation system status check for relative position.
absolute_position_ok	boolean	Navigation system status check for absolute position.
have_bottom_lock	boolean	Navigation system status check for bottom lock.

Table 8: LCM type nav_solution_t.

3.3.6 pose_stat_t

pose_stat_t is the LCM type is used to describe the position and orientation of a body in a user-chosen coordinate system. In order to use this information, the transform between the reporting device reference frame relates to the vehicle or Earth reference frame must be taken into account. Check out the table.

Variable	*Type*	*Description*
unix_time_sec	double	Unix Time stamp. The message data time stamp in seconds since January 1, 1970, 12 a.m. GMT.
count_publish	int64_t	Incremented count value for the number of times the application published this message.
count_publish_error	int64_t	Error count value for the number of times an error was encountered during message transmissions. This includes “No communications” and device errors.
position_valid	boolean	“True” if the position variables contain valid data.
position_x	double	X value of position. Coordinate system is application-dependent.
position_y	double	Y value of position. Coordinate system is application-dependent.
position_z	double	Z value of position. Coordinate system is application-dependent.
position_coordinate_system	string	String such as 'geographic' or 'relative' used to define the coordinate system that defines the position variables
attitude_valid	boolean	“True” if the attitude variables contain valid data.

Variable	*Type*	*Description*
attitude_roll_deg	double	Roll attitude in degrees.
attitude_pitch_deg	double	Pitch attitude in degrees.
attitude_heading_deg	double	Heading attitude in degrees.
sog_valid	boolean	“True” if the sog_m_sec variable contains valid speed.
sog_m_sec	double	Speed over ground in meters per second.
cog_valid	boolean	“True” if the cog_deg variable contains valid course.
cog_deg	double	Course over ground in degrees.

Table 9: LCM type pos_stat_t.

3.3.7 pressure_stat_t

pressure_stat_t is the LCM type for the system to transmit the reading from a pressure sensor to calculate depth.

Variable	*Type*	*Description*
time_unix_sec	double	Unix Time stamps. The message data time stamp in seconds since January 1, 1970, 12 p.m. GMT.
count_publish	int64_t	Incremented count value for the number of times the application published this message.
count_publish_error	int64_t	Error count value for the number of times an error was encountered during message transmissions. This includes “No communications” and device errors.
id_device	string	The device identification string. It can be used for the manufacturer and model number.
num_analogs	int32_t	The number of analog signals.
analogs[num_analogs]	analog_t	Vector of analog values. The vector contains a name and a double value.
pressure_psi	double	A variable that defines the pressure value.

Table 10: LCM type pressure_stat_t.

3.4 Configuration

The system must be configured properly to use an openINS software package. We've provided configuration files with initialized application details so the system will communicate without recompiling the program for each individual component. Those application details live in a configuration file inside the openins_config directory. The configuration file is a .yml (YAML) file. This human readable configuration file format contains the startup initialization values for the devices.

3.4.1 openins_app.yml

The openins_app.yml file is the main configuration file for the openINS package. This file contains the reference links for the openINS default details required for the package operation.

3.4.2 openins_lcm_config.yml

The openins_lcm_config.yml file contains the configuration details for LCM communications. This file provides the channel names for the corresponding devices. These subscribe channels should match what the navigation sensors are publishing.

3.4.3 openins_xforms.yml

The xforms configuration file contains the default details for the IMU, magnetic declination, depth sensor, DVL, and vehicle positional point of reference. The six degrees of freedom are the values of offset for x, y, z directions as well as phi, theta and psi angles of motion.

3.4.4 openins_state_alarms.yml

The state alarms configuration file sets the name, enumerated value, priority, threshold, delay value, type, and message of each alarm for the system. These alarms are:

STARTUP_COMPLETE

INITIALIZE COMPLETE

SYSTEM_OK

PCOMMS_LOST

The configuration settings will initialize the alarms to respond to their set levels and print the error messages defined in this file.

3.4.5 openins_task_alarms.yml

The task alarm configuration file for openINS contains the settings for the name, enumerated value, threshold, delay value, type, and message of each alarm for the system.

3.4.6 openins_task_config.yml

All specific task configurations are contained within the openINS task configuration file. Task configurations typically configure specific sensor and system configurations. All task configurations are set on startup and are not reconfigured. Below is a table describing the available task configurations:

Task Configuration	*Description*
Pconfig_gps_max_accepted_hdop	The maximum accepted HDOP value the INS will accept as a valid GPS signal.
Pconfig_dvl_dvl_db	The deadband applied to the DVL velocity signal.
Pconfig_dvl_max_accel	The maximum accepted DVL acceleration.
Pconfig_dvl_max_vel	The maximum accepted DVL velocity

Task Configuration	*Description*
Pconfig_depth_water_temperature	The temperature of the water used for the depth calculation.
Pconfig_depth_water_salinity	The salinity of the water used for the depth calculation.
Pconfig_depth_calculation_type	The depth calculation type. Available options: • 1 (UNESCO 1983, good estimation for both shallow and deep applications) • 2 (Best below 500 meters) • 3 (Simplified Saunder and Fofonoff 1976) See the Analogdepth class in openSEA for a more detailed description of the depth algorithms.
Pconfig_fuse_cmps_and_imu	Enable compass and IMU attitude fusion. “True” to enable compass and IMU fusion.
Pconfig_disable_imu_on_startup	Disable the requirement of having an IMU on startup of openINS.

Table 11: Task configurations.

4 That openINS Example Program Promised in Section 1.1

The gss_openINS_example program is a working example that's fully configured to build and send commands, allowing users to familiarize themselves with the environment prior to integrating the system.

First, you need to read the following section that walks through the building, running, and menu options available in the openINS example. The entire openINS example program source code lives in the Appendix of this document if you want to view it.

4.1 Building the openINS Example Program

Use the following steps to build an example program:

1. Open a command prompt and navigate to the folder called gss_openins_example
2. Type the following command at the prompt:

```
g++ -o gss_openins_example ./src/gss_openins_example.cpp -I./src/ -llcm -lgss_types
```

4.2 Running the Example

Once gss_openins_example builds, the program is run with the following command at the command line prompt:

```
./gss_openins_example
```

4.3 Menu Options

The commands to configure the workspace to the example program can be seen by typing 'M' for menu. The menu options are outlined in the following table. They're also explained in detail in the section below.

Command	*Result*
'M'	Displays the menu.
'D'	Displays the current navigation solution message on screen.
'P'	Displays the current openINS primary communications message.
'S'	Sets the openINS position.
'C'	Sets the magnetic declination.
'Y'	Transmits a USBL message to openINS.
'Z'	Calibrates the depth to 0.
'X'	Closes the application.

Table 12: Menu option commands and result.

4.4 Menu Functions

Knowing how the openINS application creates a dynamic positional solution with all its functions is important. The next section details the key functions listed in the table above. Remember the bit about LCM Types? You can refer to that for specific details of the ones we'll use here.

4.4.1 'Z' (publish_cal_depth())

The vehicle needs proper depth calibration when it starts in order to return accurate readings. This calibration has to happen before each dive the vehicle takes. The publish_cal_depth function will send a message to the openINS telling the system to calibrate the sensor to 0. To use the calibration function press **shift** + **Z**.

4.4.2 'D' (print_nav())

Using the print_nav function sends the current navigation solution to the display screen. This will read the complete navigation solution to the console. To use this function press **shift** + **D**.

4.4.3 'Y'(publish_usbl_data())

This function will publish a USBL message to openINS on the openINS USBL data channel, which requires new and valid data. This data is used to create a pose_stat_t type. To use this function press **shift** + **Y**. The console screen will prompt you for the desired latitude and longitude. Type in the coordinates and press **enter**.

4.4.4 'P'(print_pcomms())

The primary communications (PCOMMS) is displayed on screen with this function. Once the command is entered, the console will display the number and value of the analog and the digital values. The function is enabled by pressing **shift** + **P**.

4.4.5 'S'(publish_position_change())

This function is used to send a set positional change to openINS. You'll be prompted to enter the latitude and longitude of the new position. The openINS will create a PCOMMS message with these signals to set the

position and then prints an LCM message. To use this function press **shift** + **S**.

4.4.6 'C'(publish_set_declination())

The magnetic declination is set with this function, sending a declination message to openINS from the operator input. To use this function press **shift** + **C**.

4.5 Using lcm-spy

The navigation solution should now be producing data. Congratulations! To view the messages you'll need to call on a workspace utility called lcm-spy. Lcm-spy requires the data to be decoded into the transmitted data types for viewing. To decode the files, open a command prompt and use the cd (cd <file>) command to enter the appropriate file directory. The following steps will walk you through creating the types:

Note: This is merely an example. Individual set ups require some path modifications.

1. To make the classes, first cd into the folder with the .lcm files. The type:

   ```
   lcm-gen -j *.lcm
   ```

2. Build the classes:

   ```
   javac gss/*.java
   ```

3. Package the classes as a java 'jar' file

   ```
   jar cf gss_types.jar *.class
   ```

4. Include those 'jar' files in your environments java “CLASSPATH”. Make sure the gss_types.jar file has been created. Find the entire classpath and type it into the variable script. For example:

   ```
   export CLASSPATH=$CLASSPATH:/myhomedir/gss_openins_example_wc/src/gss/gss_types.jar
   ```

5. Once the communications type is correct, run lcm-spy by calling it explicitly from the command line:

   ```
   lcm-spy
   ```

   Lcm-spy will look something like this (double-click on a channel for the second window):

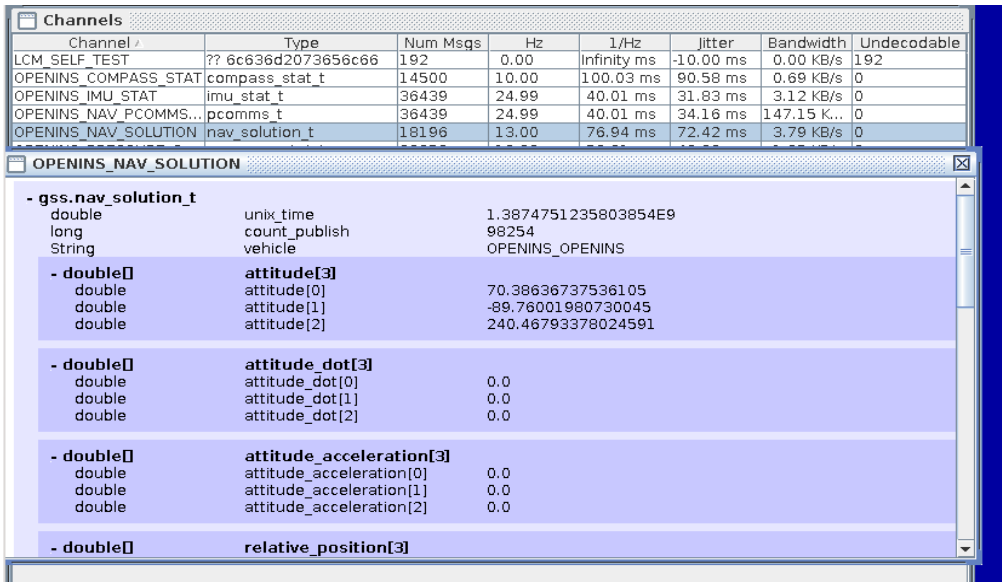

Illustration 4: LCM spy program.

5 Appendix

Up next is the verbatim block of code from the openINS example.

```
/******************************************************************************

 *   CONFIDENTIAL - PROPERTY OF GREENSEA SYSTEMS, INC.                        *
 *                                                                            *
 *   Copyright 2013 Greensea Systems, Inc.                                    *
 *   All Rights Reserved by Greensea Systems, Inc.                            *
 *                                                                            *
 *   NOTICE:  This is proprietary and confidential software.  The source      *
 *   code and intellectual property contained within this file is             *
 *   protected by one or more licenses and is NOT "free" or "open source".    *
 *   This software may not be duplicated, copied either literally or in       *
 *   spirit,  modified, or used in any way not explicitly granted in          *
 *   writing by Greensea Systems, Inc.                                        *
 *                                                                            *
 *   A copy of the governing software license for the program                 *
 *   including this file should have accompanied its distribution.  Please    *
 *   contact Greensea Systems, Inc. to receive a copy of the license.         *
 *                                                                            *
 *          Greensea Systems, Inc.                                            *
 *          10 East Main Street                                               *
 *          PO Box 959                                                        *
 *          Richmond, Vermont  05477  USA                                     *
 *          www.greenseainc.com :: info@greenseainc.com                       *
 *                                                                            *
 *                                                                            *
 ******************************************************************************/
```

```
/*****************************************************************************
 *                                                                           *
 *   This software module uses the Open Software and Equipment Architecture*
 *   (OpenSEA) library developed and maintained by Greensea Systems, Inc.  *
 *   OpenSEA is available under the terms of the LGPL license.             *
 *                                                                           *
 *   OpenSEA is a trademark of Greensea Systems, Inc. and is distributed   *
 *   by Greensea Systems, Inc. under the terms of the LGPL license.        *
 *                                                                           *
 *          Greensea Systems, Inc.                                         *
 *          10 East Main Street                                            *
 *          PO Box 959                                                     *
 *          Richmond, Vermont  05477  USA                                  *
 *          www.greenseainc.com :: info@greenseainc.com                    *
 *                                                                           *
 *****************************************************************************/
/**
  /b Overview
  OpenINS uses LCM as its primary communications path.  This program shows
  the general functionality of LCM and basic communications with OpenINS.  The
  main functionality of LCM is receiving messages and publishing messages.  This
  program gives two examples of publishing LCM messages to OpenINS; setting a
  new position and setting a magnetic declination.

  /b LCM /b General /b Philosophy
  Messages are published to a network with a specific channel name and
  received by all users subscribed to the specific channel.  The messages
  are predefined data types consisting of combinations of LCM's primitive
  types, shown below.

  /b Primitive /b Types
  int8_t  8-bit signed integer
  int16_t 16-bit signed integer
  int32_t 32-bit signed integer
  int64_t 64-bit signed integer
  float 32-bit IEEE floating point value
  double  64-bit IEEE floating point value
  string  UTF-8 string
  boolean true/false logical value
  byte  8-bit value

  LCM allows multi-dimensional arrays and constant values, for more information
  on LCM's primitive types see:
  http://lcm.googlecode.com/svn/www/reference/lcm/type_specification.html

  For detailed explanations of LCM and associated tools see LCM's project home
  page:
  https://code.google.com/p/lcm/

*/
```

```
#include <iostream>
// to handle CNTRL-C
#include <signal.h>
// LCM include
#include <lcm/lcm-cpp.hpp>
//LCM Type definitions
#include <gss/nav_solution_t.hpp>
#include <gss/pcomms_t.hpp>
#include <gss/pose_stat_t.hpp>

using namespace std;

/**
 * Create a LCM network
 */
lcm::LCM m_lcm_network;

/*
 * Navigation solution message receive channel.
 */
std::string m_subscribe_channel_nav = "OPENINS_NAV_SOLUTION";

/*
 * OpenINS PCOMMS message receive channel.
 */
std::string m_subscribe_channel_pcomms = "OPENINS_NAV_PCOMMS_TX";

/*
 * OpenINS command publish channel
 */
std::string m_openins_channel = "OPENINS_NAV_CMD";

/*
 * OpenINS USBL publish channel
 */
std::string m_openins_usbl_channel = "OPENINS_USBL_STAT";

/*
 * Storage container for the received Navigation Solution.
 */
gss::nav_solution_t m_nav_solution;

/*
 * Storage container for the received OpenINS PCOMMS message.
 */
gss::pcomms_t m_pcomms;

/**
 * A flag to indicate the program should exit.
 */
```

```
bool m_quit = false;

/**
 * Handle CNTRL-C
 * \param signo
 * \return VOID
 */
static void sigcatch_SIGINT(int signo);

/*
 * Menu function for the gss_openins_example program.
 * \return void
 */
void display_menu();

/**
 *  Manages user inputs.
 *  \return VOID
 */
void get_input_cmd();

/**
 * Initialize the navigation solution data type. C++ does not initialize
 * the LCM types, so we must explicitly initialize each member or the type.
 * \return navigation solution data type
 */
gss::nav_solution_t init_soln_data();

/**
 * Initialize the PCOMMS data type. C++ does not initialize
 * the LCM types, so we must explicitly initialize each member or the type.
 * \return pcomms data type
 */
gss::pcomms_t init_pcomms_data();

/**
 * Initialize the pose data type. C++ does not initialize
 * the LCM types, so we must explicitly initialize each member or the type.
 * \return pose data type
 */
gss::pose_stat_t init_pose_data();

/**
 * Print the current navigation solution message.
 *  \return VOID
 */
void print_nav();

/**
 * Print the current OpenINS PCOMMS message.
```

```
 *  \return VOID
 */
void print_pcomms();

/**
 * Check to see if there are lcm messages to be handled
 * \param file_descriptor
 * \return bool True if there are received messages
 */
bool have_rx_messages(int file_descriptor);

/**
 * Return a double corresponding to the number of seconds since
 * Midnight January 1, 1970 GMT
 * \return unix time stamp as a double
 */
double get_unix_timestamp();

/**
 * Publish a Set Position message to OpenINS.  This function will
 * ask the user for the desired latitude and longitude, then will create
 * a PCOMMS message with the correct signals to set the OpenINS position, and
 * will finally publish the LCM message.
 * \return void
 */
void publish_position_change();

/**
 * Publish a Set Magnetic Declination message to OpenINS.  This function will
 * ask the user for the desired magnetic declination, then will create
 * a PCOMMS message with the correct signals to set the OpenINS declination, and
 * will finally publish the LCM message.
 * \return void
 */
void publish_set_declination();

/**
 * Publish a USBL data message to OpenINS.   OpenINS uses generic data types to
 * get different sensors data.  USBL is input with a pose_stat_t.  When
 * implementing this message with a sensor it is important to only publish
 * new valid data, OpenINS will interpret any data received as such.  This
 * function will ask the user for the desired latitude and longitude, then
 * create a pose_stat_t with the input latitude and longitude, and finally will
 * publish the data to OpenINS on the USBL data channel.
 * \return void
 */
void publish_usbl_data();

/**
 * Publish a calibrate depth message to OpenINS.  This function sends OpenINS
 * the signal to calibrate the depth to 0.  Depth calibrations should be
```

```
 * performed before each dive on the deck, ensuring your depth reading is
 * as accurate as possible.
 * \return void
 */
void publish_cal_depth();

//////////////////////////////////////////////////////////////////////
/**
 * Handle CNTRL-C
 * \param signo
 * \return VOID
 */
static void sigcatch_SIGINT(int signo)
{
  cout << "I FOUND SIGINT - " << signo << endl;
  m_quit = true;
}

//////////////////////////////////////////////////////////////////////

/**
 * \class Receive_callback
 * \brief Callback class for incoming navigation solution messages
 */
class Receive_nav_callback
{
public:
  /**
   * Desturctor
   */
  ~Receive_nav_callback() {}

  /**
   * On message received gets the data from the subscribe channel and
   * processes the data accordingly.
   * \param rbuf LCM receive buffer
   * \param chan LCM subscribe channel
   * \param msg LCM data type containing the message information
   * \return VOID
   */
  void On_message_received(const lcm::ReceiveBuffer* rbuf,
      const std::string& chan,
      const gss::nav_solution_t * msg)
  {
    //Set our navigation solution container to the received message.
    m_nav_solution = *msg;
  }
};
//////////////////////////////////////////////////////////////////////

/**
```

```
 * \class Receive_callback
 * \brief Callback class for incoming OpenINS PCOMMS messages
 */
class Receive_pcomms_callback
{
public:
  /**
   * Desturctor
   */
  ~Receive_pcomms_callback() {}

  /**
   * On message received gets the data from the subscribe channel and
   * processes the data accordingly.
   * \param rbuf LCM receive buffer
   * \param chan LCM subscribe channel
   * \param msg LCM data type containing the message information
   * \return VOID
   */
  void On_message_received(const lcm::ReceiveBuffer* rbuf,
      const std::string& chan,
      const gss::pcomms_t * msg)
  {
    m_pcomms = *msg;
  }
};

//////////////////////////////////////////////////////////////////////

/*
 * Menu function for the gss_openins_example program.
 * \return void
 */
void display_menu()
{
  printf("\n************************************************\n\n");
  printf("************************************************\n");
  printf("Commands for use:\n");
  printf("    'M','m' = Display this menu\n");
  printf("    'D','d' = Display the current navigation solution message\n");
  printf("    'P','p' = Display the current OpenINS PCOMMS message\n");
  printf("    'S','s' = Set OpenINS position\n");
  printf("    'C','c' = Set OpenINS magnetic declination\n");
  printf("    'Y','y' = Transmit a USBL message to OpenINS\n");
  printf("    'Z','z' = Calibrate the Depth to 0\n");
  printf("    'X','x',CTRL+C = Close application\n");
  printf("************************************************\n\n");
}

//////////////////////////////////////////////////////////////////////
```

```
/**
 *  Manages user inputs.
 *  \return VOID
 */
void get_input_cmd()
{
  char r = 'U';
  cin.clear();
  cin >> r;
  if(r=='M' || r=='m')
  {
    display_menu();
    return;
  }

  if(r=='D' || r=='d')
  {
    print_nav();
    return;
  }

  if(r=='P' || r=='p')
  {
    print_pcomms();
    return;
  }

  if(r=='S' || r=='s')
  {
    publish_position_change();
    return;
  }

  if(r=='C' || r=='c')
  {
    publish_set_declination();
    return;
  }

  if(r=='Y' || r=='y')
  {
    publish_usbl_data();
    return;
  }

  if(r=='Z' || r=='z')
  {
    publish_cal_depth();
    return;
  }
```

```
  if(r=='X' || r=='x')
  {
    m_quit = true;
    return;
  }

  return;
}

////////////////////////////////////////////////////////////////////////
/**
 * Initialize the navigation solution data type. C++ does not initialize
 * the LCM types, so we must explicitly initialize each member or the type.
 * \return  navigation solution data type
 */
gss::nav_solution_t init_soln_data()
{
 gss::nav_solution_t nav_out;

 nav_out.absolute_position[0] = 0;
 nav_out.absolute_position[1] = 0;
 nav_out.absolute_position[2] = 0;
 nav_out.absolute_position_ok = false;
 nav_out.altitude_above_bottom = 0;
 nav_out.altitude_ok = false;
 nav_out.attitude[0] = 0;
 nav_out.attitude[1] = 0;
 nav_out.attitude[2] = 0;
 nav_out.attitude_acceleration[0] = 0;
 nav_out.attitude_acceleration[1] = 0;
 nav_out.attitude_acceleration[2] = 0;
 nav_out.attitude_dot[0] = 0;
 nav_out.attitude_dot[1] = 0;
 nav_out.attitude_dot[2] = 0;
 nav_out.attitude_ok = false;
 nav_out.course_over_ground = 0;
 nav_out.depth = 0;
 nav_out.depth_ok = false;
 nav_out.dvl_error = 0;
 nav_out.have_bottom_lock = false;
 nav_out.imu_error = 0;
 nav_out.initial_lonlat_fix[0] = 0;
 nav_out.initial_lonlat_fix[1] = 0;
 nav_out.last_lonlat_fix[0] = 0;
 nav_out.last_lonlat_fix[1] = 0;
 nav_out.relative_acceleration[0] = 0;
 nav_out.relative_acceleration[1] = 0;
 nav_out.relative_acceleration[2] = 0;
 nav_out.relative_position[0] = 0;
 nav_out.relative_position[1] = 0;
 nav_out.relative_position[2] = 0;
```

```
 nav_out.relative_position_dot[0] = 0;
 nav_out.relative_position_dot[1] = 0;
 nav_out.relative_position_dot[2] = 0;
 nav_out.relative_position_ok = false;
 nav_out.speed_ok = false;
 nav_out.speed_over_ground = 0;
 nav_out.unix_time = 0;
 nav_out.count_publish = 0;
 nav_out.vehicle = "";
 return nav_out;
}

////////////////////////////////////////////////////////////////////////

/**
 * Initialize the PCOMMS data type. C++ does not initialize
 * the LCM types, so we must explicitly initialize each member or the type.
 * \return pcomms data type
 */
gss::pcomms_t init_pcomms_data()
{
  gss::pcomms_t pcomms_out;
  pcomms_out.num_analogs = 0;
  pcomms_out.num_digitals = 0;
  pcomms_out.sender_id = "";
  pcomms_out.unix_time = 0;
  pcomms_out.analogs.clear();
  pcomms_out.digitals.clear();
  return pcomms_out;
}

////////////////////////////////////////////////////////////////////////

/**
 * Initialize the pose data type. C++ does not initialize
 * the LCM types, so we must explicitly initialize each member or the type.
 * \return pose data type
 */
gss::pose_stat_t init_pose_data()
{
  gss::pose_stat_t pose_out;
  pose_out.id_device = "";
  pose_out.time_unix_sec = 0;
  pose_out.num_analogs = 0;
  pose_out.analogs.clear();
  pose_out.count_publish = 0;
  pose_out.count_publish_error = 0;
  pose_out.position_coordinate_system = "";

  pose_out.attitude_valid = false;
  pose_out.attitude_roll_deg = 0;
```

```
  pose_out.attitude_pitch_deg = 0;
  pose_out.attitude_heading_deg = 0;

  pose_out.position_valid = false;
  pose_out.position_x = 0;
  pose_out.position_y = 0;
  pose_out.position_z = 0;

  pose_out.cog_valid = false;
  pose_out.cog_deg = 0;
  pose_out.sog_valid = false;
  pose_out.sog_m_sec = 0;
  return pose_out;
}

////////////////////////////////////////////////////////////////////

/**
 * Print the current navigation solution.
 *  \return VOID
 */
void print_nav()
{
  printf("\nCurrent Navigation Solution:\n");
  printf("unix time = %f\n",m_nav_solution.unix_time);
  printf("vehicle = %s\n",m_nav_solution.vehicle.c_str());
  printf("number of published messages = %lld\n",m_nav_solution.count_publish);

  //Absolute Position Data
  printf("absolute position ok = %s\n",m_nav_solution.absolute_position_ok ?
"True":"False");
  printf("longitude = %f\n",m_nav_solution.absolute_position[0]);
  printf("latitude = %f\n",m_nav_solution.absolute_position[1]);
  printf("depth = %f\n",m_nav_solution.absolute_position[2]);

  //Attitude Data
  printf("attitude ok = %s\n",m_nav_solution.attitude_ok? "True":"False");
  printf("roll = %f\n",m_nav_solution.attitude[0]);
  printf("pitch = %f\n",m_nav_solution.attitude[1]);
  printf("heading = %f\n",m_nav_solution.attitude[2]);
  printf("roll velocity = %f\n",m_nav_solution.attitude_dot[0]);
  printf("pitch velocity = %f\n",m_nav_solution.attitude_dot[1]);
  printf("heading velocity = %f\n",m_nav_solution.attitude_dot[2]);
  printf("roll acceleration = %f\n",m_nav_solution.attitude_acceleration[0]);
  printf("pitch acceleration = %f\n",m_nav_solution.attitude_acceleration[1]);
  printf("heading acceleration = %f\n",m_nav_solution.attitude_acceleration[2]);

  //Relative Position Data
  printf("relative position ok = %s\n",m_nav_solution.relative_position_ok ?
"True":"False");
  printf("northing = %f\n",m_nav_solution.relative_position[0]);
```

```
  printf("easting = %f\n",m_nav_solution.relative_position[1]);
  printf("downing = %f\n",m_nav_solution.relative_position[2]);
  printf("northing velocity = %f\n",m_nav_solution.relative_position_dot[0]);
  printf("easting velocity = %f\n",m_nav_solution.relative_position_dot[1]);
  printf("downing velocity = %f\n",m_nav_solution.relative_position_dot[2]);
  printf("northing acceleration = %f\n",m_nav_solution.relative_acceleration[0]);
  printf("easting acceleration = %f\n",m_nav_solution.relative_acceleration[1]);
  printf("downing acceleration = %f\n",m_nav_solution.relative_acceleration[2]);

  //Depth Data
  printf("depth ok = %s\n",m_nav_solution.depth_ok ? "True":"False");
  printf("depth = %f\n",m_nav_solution.depth);

  //Altitude Data
  printf("altitude ok = %s\n",m_nav_solution.altitude_ok ? "True":"False");
  printf("altitude above bottom = %f\n",m_nav_solution.altitude_above_bottom);
  printf("have bottom lock = %s\n",m_nav_solution.have_bottom_lock ?
"True":"False");

  //General data
  printf("speed ok = %s\n",m_nav_solution.speed_ok ? "True":"False");
  printf("speed over ground = %f\n",m_nav_solution.speed_over_ground);
  printf("course over ground = %f\n",m_nav_solution.course_over_ground);
  printf("imu error = %f\n",m_nav_solution.imu_error);
  printf("dvl error = %f\n",m_nav_solution.dvl_error);
  printf("initial longitude = %f\n",m_nav_solution.initial_lonlat_fix[0]);
  printf("initial latitude = %f\n",m_nav_solution.initial_lonlat_fix[1]);
  printf("last longitude = %f\n",m_nav_solution.last_lonlat_fix[0]);
  printf("last latitude = %f\n",m_nav_solution.last_lonlat_fix[1]);

}

//////////////////////////////////////////////////////////////////////

/**
 * Print the current OpenINS PCOMMS message.
 *  \return VOID
 */
void print_pcomms()
{
  printf("\nCurrent OpenINS PCOMMS:\n");
  printf("Received %d Digital Values:\n",m_pcomms.num_digitals);
  //Loop through all of the digital values stored in the PCOMMS message
  for(int indx = 0; indx < m_pcomms.num_digitals; indx++)
  {
    //Print out the name and value each digital value
    printf("%s: %s\n",m_pcomms.digitals[indx].name.c_str(),
        m_pcomms.digitals[indx].value ? "True":"False");
  }

  printf("Received %d Analog Values:\n",m_pcomms.num_analogs);
```

```
  //Loop through all of the digital values stored in the PCOMMS message
  for(int indx = 0; indx < m_pcomms.num_analogs; indx++)
  {
    //Print out the name and value each digital value
    printf("%s: %f\n",m_pcomms.analogs[indx].name.c_str(),
        m_pcomms.analogs[indx].value);
  }

}

////////////////////////////////////////////////////////////////////////

/**
 * Check to see if there are LCM messages to be handled
 * \param file_descriptor
 * \return bool True if there are received messages
 */
bool have_rx_messages(int file_descriptor)
{
  fd_set file_read_set;
  FD_ZERO(&file_read_set);
  FD_SET(file_descriptor, &file_read_set);
  struct timeval timeout;
  timeout.tv_sec = 0; // zero timeout makes select return immediately
  timeout.tv_usec = 0;
  int retval = select(file_descriptor+1, &file_read_set, (fd_set *)0,
      (fd_set *)0, &timeout);
  return ( retval > 0 && FD_ISSET(file_descriptor, &file_read_set) );
}

////////////////////////////////////////////////////////////////////////

/**
 * Return a double corresponding to the number of seconds since
 * Midnight January 1, 1970 GMT
 * \return unix time stamp as a double
 */
double get_unix_timestamp()
{
  timespec curr_time;
  if (clock_gettime(CLOCK_REALTIME, &curr_time) < 0)
    return 0.0;
  else // we read time successfully
    return (curr_time.tv_sec + curr_time.tv_nsec * 1e-9);
}

////////////////////////////////////////////////////////////////////////

/**
 * Publish a Set Position message to OpenINS.  This function will
 * ask the user for the desired latitude and longitude, then will create
```

```
 * a PCOMMS message with the correct signals to set the OpenINS position, and
 * will finally publish the LCM message.
 * \return void
 */
void publish_position_change()
{
  printf("Position Change Request.\n");
  printf("Enter Your Latitude\n");
  double lat = 0;
  cin.clear();
  cin >> lat;
  printf("Enter Your Longitude\n");
  double lon = 0;
  cin.clear();
  cin >> lon;

  //Initialize the PCOMMS message.
  gss::pcomms_t pcomms = init_pcomms_data();
  //PCOMMS is made of one vector of analog_t types and one vector
  //of digital_t types.  The analog_t type has a double value and a string name
  //to describe the value.  The digital_t type has a boolean value and a string
  //name to describe the value.

  //Create a temporary analog_t
  gss::analog_t temp_anlg;
  //Use the User Set latitude signal to populate your new latitude
  temp_anlg.name = "U_INIT_LAT";
  //Set the new latitude to the value of the analog_t
  temp_anlg.value = lat;
  //Add the temporary analog_t to PCOMMS analogs vector
  pcomms.analogs.push_back(temp_anlg);

  //Use the User Set longitude signal to populate your new longitude
  temp_anlg.name = "U_INIT_LON";
  //Set the new longitude to the value of the analog_t
  temp_anlg.value = lon;
  //Add the temporary analog_t to PCOMMS analogs vector
  pcomms.analogs.push_back(temp_anlg);
  //Get the size of our analogs vector and set it to the num_analogs signal
  pcomms.num_analogs = pcomms.analogs.size();

  //Create a temporary digital_t
  gss::digital_t temp_dig;
  //We need to tell OpenINS that we want to initialize our new position
  //Use the user initialize position name for position initialization
  temp_dig.name ="U_INIT_POS";
  //Set the value to true
  temp_dig.value = true;
  //Add the new digital_t to the vector of digitals
  pcomms.digitals.push_back(temp_dig);
  //Get the size of our digitals vector and set it to the num_digitals signal
```

```
  pcomms.num_digitals = pcomms.digitals.size();
  //Set the Sender ID to example
  pcomms.sender_id = "EXAMPLE";
  //publish the message with the channel name and build message
  m_lcm_network.publish(m_openins_channel,&pcomms);
}
////////////////////////////////////////////////////////////////////

/**
 * Publish a Set Magnetic Declination message to OpenINS.  This function will
 * ask the user for the desired magnetic declination, then will create
 * a PCOMMS message with the correct signals to set the OpenINS declination, and
 * will finally publish the LCM message.
 * \return void
 */
void publish_set_declination()
{
  printf("Set Magnetic Declination.\n");
  printf("Enter Your Magnetic Declination\n");
  double dec = 0;
  cin.clear();
  cin >> dec;

  //Initialize the PCOMMS message.
  gss::pcomms_t pcomms = init_pcomms_data();
  //PCOMMS is made of one vector of analog_t types and one vector
  //of digital_t types.  The analog_t type has a double value and a string name
  //to describe the value.  The digital_t type has a boolean value and a string
  //name to describe the value.

  //Create a temporary analog_t
  gss::analog_t temp_anlg;
  //Use the User Set latitude signal to populate your new latitude
  temp_anlg.name = "U_MAG_DEC";
  //Set the new latitude to the value of the analog_t
  temp_anlg.value = dec;
  //Add the temporary analog_t to PCOMMS analogs vector
  pcomms.analogs.push_back(temp_anlg);
  //Get the size of our analogs vector and set it to the num_analogs signal
  pcomms.num_analogs = pcomms.analogs.size();
  //publish the message with the channel name and build message
  m_lcm_network.publish(m_openins_channel,&pcomms);
}

////////////////////////////////////////////////////////////////////

/**
 * Publish a USBL data message to OpenINS.   OpenINS uses generic data types to
 * get different sensors data.  USBL is input with a pose_stat_t.  When
 * implementing this message with a sensor it is important to only publish
 * new valid data, OpenINS will interpret any data received as such.  This
```

```
 * function will ask the user for the desired latitude and longitude, then
 * create a pose_stat_t with the input latitude and longitude, and finally will
 * publish the data to OpenINS on the USBL data channel.
 * \return void
 */
void publish_usbl_data()
{
  printf("Publish a USBL Data Message.\n");
  printf("Enter Your Latitude\n");
  double lat = 0;
  cin.clear();
  cin >> lat;
  printf("Enter Your Longitude\n");
  double lon = 0;
  cin.clear();
  cin >> lon;

  //Initialize the pose data
  gss::pose_stat_t pose_out = init_pose_data();
  //X is Latitude
  pose_out.position_x = lat;
  //Y is Longitude
  pose_out.position_y = lon;
  //Z is unused.
  //Set the position valid flag to true.
  pose_out.position_valid = true;
  //Set the current Unix time to the time variable
  pose_out.time_unix_sec = get_unix_timestamp();
  //Set the device name and coordinate system
  pose_out.id_device = "EXAMPLE";
  pose_out.position_coordinate_system = "ABSOLUTE";
  //publish the message with the channel name and build message
  m_lcm_network.publish(m_openins_usbl_channel,&pose_out);
}

////////////////////////////////////////////////////////////////////////

/**
 * Publish a calibrate depth message to OpenINS.  This function sends OpenINS
 * the signal to calibrate the depth to 0.  Depth calibrations should be
 * performed before each dive on the deck, ensuring your depth reading is
 * as accurate as possible.
 * \return void
 */
void publish_cal_depth()
{
  //Initialize the PCOMMS message.
  gss::pcomms_t pcomms = init_pcomms_data();
  //PCOMMS is made of one vector of analog_t types and one vector
  //of digital_t types.  The analog_t type has a double value and a string name
  //to describe the value.  The digital_t type has a boolean value and a string
```

```
  //name to describe the value.

  //Create a temporary digital_t
  gss::digital_t temp_dig;
  //We need to tell OpenINS that we want to calibrate our depth to 0
  //Use the user calibrate depth signal
  temp_dig.name ="U_CAL_DEPTH";
  //Set the value to true
  temp_dig.value = true;
  //Add the new digital_t to the vector of digitals
  pcomms.digitals.push_back(temp_dig);
  //Get the size of our digitals vector and set it to the num_digitals signal
  pcomms.num_digitals = pcomms.digitals.size();
  //Set the Sender ID to example
  pcomms.sender_id = "EXAMPLE";
  //publish the message with the channel name and build message
  m_lcm_network.publish(m_openins_channel,&pcomms);
}

////////////////////////////////////////////////////////////////////////

int main() {

  printf("gss_openins_example Example LCM Application\n");
  printf("Developed by Greensea Systems, Inc.\n");
  printf("Greensea Systems, Inc. ::  www.greenseainc.com\n\n");

  //Install the signal handler for SIGINT
  struct sigaction act;
  act.sa_handler = sigcatch_SIGINT;
  act.sa_flags = 0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGINT, &act, NULL);

  //Initialize the LCM member types
  m_nav_solution = init_soln_data();
  m_pcomms = init_pcomms_data();

  //Ensure our LCM network was initialized correctly
  if(!m_lcm_network.good())
  {
    printf("LCM Network Did Not Initialize Correctly, Exiting Program!\n");
    return EXIT_FAILURE;
  }

  //Subscribe our navigation solution callback class to our LCM network
  Receive_nav_callback handlerObject_nav;
  m_lcm_network.subscribe(m_subscribe_channel_nav,  // command input channel
      &Receive_nav_callback::On_message_received,   // class callback function
      &handlerObject_nav);                          // class instance
```

```
  //Subscribe our OpenINS PCOMMS callback class to our LCM network
  Receive_pcomms_callback handlerObject_pcomms;
  m_lcm_network.subscribe(m_subscribe_channel_pcomms,  // command input channel
      &Receive_pcomms_callback::On_message_received,   // class callback function
      &handlerObject_pcomms);                          // class instance

  //All of the functions rely on communications with OpenINS so
  //wait here until we receive our first message.
  printf("Waiting for OpenINS Communications...\n");
  //LCM provides a blocking message handling function.  This function will
  //only return once a LCM message is received from the previously subscribed
  //channels.
  m_lcm_network.handle();
  //If CNTRL-C is hit just return here
  if(m_quit)return EXIT_SUCCESS;
  //If we make it here we have received a message from OpenINS
  printf("Established OpenINS Communications...\n");

  //Display the users menu
  display_menu();

  //As soon as we get our first OpenINS message start the Main loop.
  while(!m_quit)
  {
    //Check if we have received any LCM messages.
    //The LCM handle() function is a blocking function, you can use our
    //have_rx_messages(int) function to check if there are messages before
    //calling the handle() function, essentially creating a non-blocking
    //handle function.
    //Also we want to loop through all of the received messages to ensure
    //we get the most recent messages.
    while(have_rx_messages(m_lcm_network.getFileno()))
      m_lcm_network.handle();

    //Get the input command for the user
    get_input_cmd();

    //Display the users menu
    display_menu();
  }
        return EXIT_SUCCESS;
}

////////////////////////////////////////////////////////////////////
//END OF FILE
```

End Page