**NEPI-Bot - Cloud**

Interface Control Document

*(NEPI-Bot Architectural Design Specification)*

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Author** | **Changes** |
| Draft-A | 01.Feb.2019 | John benJohn | Initial Draft. |
| Draft-B | 01.Mar.2019 | John benJohn | March updates; Bit-Packing; Config; PIPO. |
| Draft-C | 01.Apr.2019 | John benJohn | Apr updates; Bot-Comm/Iridium; logging; corrections. |
| Draft-D | 17.May.2019 | John benJohn  Kevin Ramada | May updates; includes 5/1 release status/information.  Latest Bot-Comm specifications; Iridium comms management. |
| 1.0 | 15.Jun.2019 | John benJohn | Official Document Release |

**Table of Contents**

Contents

[Purpose 8](#_Toc8805881)

[Reference Documents 8](#_Toc8805882)

[Program Architecture 9](#_Toc8805883)

[The Ocean Float Program 9](#_Toc8805884)

[The NumSDK Application 9](#_Toc8805885)

[The NEPI-Bot Application 9](#_Toc8805886)

[The NEPI-Portal Application 9](#_Toc8805887)

[NEPI-Bot General Architecture 10](#_Toc8805888)

[Introduction 10](#_Toc8805889)

[NEPI-Bot Application Design 11](#_Toc8805890)

[Bot-Recv Subsystem (*botrecv.py*) 11](#_Toc8805891)

[Bot-Send Subsystem (*botsend.py*) 12](#_Toc8805892)

[Bot-Defs Definitions File (*botdefs.py*) 12](#_Toc8805893)

[Bot-Cfg Class Library (*botcfg.py*) 12](#_Toc8805894)

[Bot-Log Class Library (*botlog.py*) 12](#_Toc8805895)

[Bot-DB Class Library (*botdb.py*) 12](#_Toc8805896)

[Bot-PIPO Class Library (*botpipo.py*) 12](#_Toc8805897)

[Bot-Msg Class Library (*botmess.py*) 13](#_Toc8805898)

[Bot-Comm Class Library (*botcomm.py*) 13](#_Toc8805899)

[Bot-Help Library (*bothelp.py*) 13](#_Toc8805900)

[NEPI-Bot Application Configuration 14](#_Toc8805901)

[Introduction 14](#_Toc8805902)

[The NEPI-Bot Configuration File (*config.json*) 14](#_Toc8805903)

[Internal Defaults 18](#_Toc8805904)

[NEPI-Bot Debugging/Logging (the *botlog.py* Class Library) 19](#_Toc8805905)

[Introduction 19](#_Toc8805906)

[Debugging Mode 19](#_Toc8805907)

[Logging Mode 19](#_Toc8805908)

[Class Instantiation 19](#_Toc8805909)

[Class Initialization 20](#_Toc8805910)

[Configuration Parameters 20](#_Toc8805911)

[Primary Output Method (*log.track()*) 22](#_Toc8805912)

[Error Output Method (*log.errtrack()*) 22](#_Toc8805913)

[Communications Manager (*botcomm.py*) 23](#_Toc8805914)

[Introduction 23](#_Toc8805915)

[Communication Object Instantiation 23](#_Toc8805916)

[Communications Connection 23](#_Toc8805917)

[Downlink Message Processing 23](#_Toc8805918)

[Uplink Message Processing 23](#_Toc8805919)

[Message Acknowledgement 23](#_Toc8805920)

[Communications Termination 24](#_Toc8805921)

[Bot-Recv Subsystem (the NEPI-Bot ‘Downlink’ Manager: *botrecv.py*) 25](#_Toc8805922)

[Introduction 25](#_Toc8805923)

[Assumptions 25](#_Toc8805924)

[Subsystem Configuration 25](#_Toc8805925)

[Control and Configuration Message Retrieval 25](#_Toc8805926)

[Control and Configuration Message Processing 26](#_Toc8805927)

[Subsystem Housekeeping 27](#_Toc8805928)

[Bot-Send Subsystem (the NEPI-Bot Uplink Manager: *botsend.py*) 28](#_Toc8805929)

[Introduction 28](#_Toc8805930)

[Assumptions 28](#_Toc8805931)

[Subsystem Configuration 29](#_Toc8805932)

[Status and Data Retrieval 29](#_Toc8805933)

[Persistent Storage (Database Manager: *botdb.py*) 31](#_Toc8805934)

[Introduction 31](#_Toc8805935)

[Database Instantiation 31](#_Toc8805936)

[Admin Table (‘*admin’*) 31](#_Toc8805937)

[Status Table (‘*status’*) 34](#_Toc8805938)

[Data Product Table (‘*meta’*) 36](#_Toc8805939)

[Data Product Prioritization (PIPO Manager: *botpipo.py*) 40](#_Toc8805940)

[Introduction 40](#_Toc8805941)

[Status and Data Product Consumption 40](#_Toc8805942)

[Data Product Prioritization 40](#_Toc8805943)

[Upload Message Buffer Creation 41](#_Toc8805944)

[Data Product Selection 41](#_Toc8805945)

[Message Packetization 41](#_Toc8805946)

[Data Product Purging 42](#_Toc8805947)

[Uplink Message Buffer Packaging (Message Manager: *botmsg.py*) 43](#_Toc8805948)

[Introduction 43](#_Toc8805949)

[Message Creation 43](#_Toc8805950)

[Request Messages 43](#_Toc8805951)

[Request Message Bit-Packing 43](#_Toc8805952)

[Status Messages 43](#_Toc8805953)

[Status Message Bit-Packing 45](#_Toc8805954)

[Data Messages 47](#_Toc8805955)

[Data Message Bit-Packing 47](#_Toc8805956)

[Message Signing 49](#_Toc8805957)

[Message Checksum 49](#_Toc8805958)

[Message Compression 49](#_Toc8805959)

[Message Encryption 49](#_Toc8805960)

[Housekeeping 50](#_Toc8805961)

[Message Delivery (Communications Manager: *botcomm.py*) 50](#_Toc8805962)

[Introduction 50](#_Toc8805963)

[Communications Connection 50](#_Toc8805964)

[Inbound Messaging 50](#_Toc8805965)

[Outbound Messaging 51](#_Toc8805966)

[Message Acknowledgement 51](#_Toc8805967)

[Communications Termination 51](#_Toc8805968)

[Housekeeping 51](#_Toc8805969)

**List of Figures**

[Figure 1: The NEPI-Bot Application with Supporting Subsystems 10](#_Toc8805970)

[Figure 2: Sample JSON-formatted Bot Configuration File (config.json) with Defaults 18](#_Toc8805971)

[Figure 3: Sample Code Depicting Instantiation of BotLog Class Object 19](#_Toc8805972)

[Figure 4: Output Snippet Showing Initialization of 1) Console Debugging and 2) File Logging 20](#_Toc8805973)

[Figure 5: Sample Code Snippet with Anticipated Debugging/Logging Output 21](#_Toc8805974)

[Figure 6: Sample Code Snippet Demonstrating Debugging/Logging Level 9 22](#_Toc8805975)

[Figure 7: Bot-Send List of “Data Folders” in the “Data Directory” (created by NumSDK) 29](#_Toc8805976)

[Figure 8: Sample Status and Data Product Files in a Typical "Data Folder" 30](#_Toc8805977)

[Figure 9: Bot-Send Processing of the Data Product File Set 30](#_Toc8805978)

[Figure 10: General Layout of a Bot-Mess Uplink Status Message. 43](#_Toc8805979)

[Figure 11: Layout of the “Msg Header” om the Bot-Mess Uplink Status Message 43](#_Toc8805980)

**List of Tables**

[Table 1: The NEPI-Bot Configuration File (JSON Formatted) 18](#_Toc8805981)

[Table 2: General Bit-Mapped Format of the Downlink Message 26](#_Toc8805982)

[Table 3: Interface File List (Reprised from the NumurusSDK - NEPI-Bot ICD). 27](#_Toc8805983)

[Table 4: The 'admin' Table Schema (with Configuration File Mappings) 33](#_Toc8805984)

[Table 5: The ‘status’ Table Schema (with Status File Mappings) 36](#_Toc8805985)

[Table 6: The ‘data’ Table Schema (with Data File Mappings) 39](#_Toc8805986)

[Table 7: Bit Map of the Status Message "Msg Header" 44](#_Toc8805987)

[Table 8: NumSDK Status Record to Cloud Message Bit-Packing Map 47](#_Toc8805988)

[Table 9: First 4 bytes (32-bits) for ALL Uplink Message "Data Segments" 48](#_Toc8805989)

[Table 10: Next 8 bytes (64-bits) for ALL Uplink Message "Data Segments” 49](#_Toc8805990)

[Table 11: Bytes 12+ in Data Segment when "std" Data Product Information is included in the Segment 49](#_Toc8805991)

**NEPI-Bot - Cloud**

**Interface Control Document**

**(NEPI-Bot Architectural Design Specification)**

# Purpose

This document specifies the functional specifications for **NEPI-Bot**, an on-board Application running on the Numurus Float Device. This “NEPI-Bot - Cloud ICD” is intended to provide adequate interface flexibility to support various projects and programs while covering the entire interface for the Ocean-of-Things (OoT) Program. In this document’s detail are architectural and design considerations that: 1) confirm the interface between the Numurus SDK and NEPI-Bot, 2) explain the NEPI-Bot Application and its various subsystems and components, and 3) define the interface(s) for controlling NEPI-Bot communication with “the Cloud” via Iridium SDB, Ethernet, Bluetooth/Wi-Fi, RS-232, and/or other communication technologies as determined by Numurus.

# Reference Documents

This “NEPI-Bot - Cloud ICD” is related to several additional documents, access to which may be granted on request. Among other useful reference documents are the following:

1)  *Float and Data Attributes Report – Concept Phase* (2018.Dec.31; Jason Seawall)

This report covers current concept float and data attributes related to Numurus’ Ocean Float Program TA-1 Phase1 development effort and is intended to support TA-2 teams’ prediction model concept development. The content in this report is based on Numurus latest concept design efforts and covers both the physical system attributes, collection & processing attributes, information transport details, as well as data and metadata publish and storage formats. In addition to the actual attributes, this document covers data creation, transition, and publish scheduling.

2) *NumSDK – NEPI-Bot Interface Control Document* (2019.Jan.03 Draft C; Josh Maximoff)

This document specifies the interface between the NumSDK (ROS-based application set) and the NEPI-Bot (on-device cloud server), designated hereafter as the Interface.

3) *Iridium Short Burst Data Service Developers Guide* (2017.Nov; MAN0025 Release 3.2)

The purpose of this document is to provide technical and operational information sufficient for an Iridium Value Added Reseller or Value-Added Manufacturer to be able to develop an integrated data application that utilizes Iridium’s Short Burst Data Service (SBD). Additional information will be required by the developer for the AT Commands to be utilized with the transceiver selected for use with SBD. An overview of the satellite network is provided as well as descriptions of the terminal equipment and the end to end communications protocol for SBD. This document is intended for use by technical personnel and assumes a reasonable level of technical skill and familiarity with satellite and/or wireless data applications.

4) *Iridium ISU AT Command Reference* (2014.Aug.25; MAN0009 Version 5)

This document is intended as a reference guide to the usage of the AT command set for the Iridium TM/SM subscriber unit. This document only applies to the Motorola satellite series. The intended audience for this document is the field test engineers, product and intelligent peripheral developers.

# Program Architecture

## The Ocean Float Program

Numurus’ **Ocean Float Program** is comprised of the following primary components: 1) Floats (*i.e.*, buoy-like devices), each with a suite of sensors, controlled by two primary, on-board Applications, NumSDK and NEPI-Bot, and 2) the NEPI Portal, a Web Application, handled by a RESTful web service layer, known as the NEPI API.

When deployed, the Floats provide Status and Data Product information (captured from their sensors) which is, then, prioritized, compacted, signed, check-summed, encrypted, and periodically sent to the “the Cloud” where it can be visualized in a variety of ways.

## The NumSDK Application

The **NumSDK** Application (also referred to as “the Numurus SDK”), comprised of various subsystems and components, is responsible for sampling the Float’s sensors, pipeline-processing the Data Product information, and storing the accumulated “Status and Data Product” records in such a way as to make them readily available to the NEPI-Bot Application. [See the “*NumSDK – NEPI-Bot Interface Control Document*” for detailed information regarding this interface.]

## The NEPI-Bot Application

The **NEPI-Bot** Application resides on the Float Device and, through its two primary subsystems, Bot-Recv and Bot-Send, provides the sole interface between the Numurus SDK (also on the Float) and the NEPI-Portal (residing in “the Cloud”). Bot-Recv receives new configuration, control, and alarm messages from “the Cloud,” acting immediately on those targeted toward NEPI-Bot itself and passing others on to the Numurus SDK for subsequent processing. Bot-Send archives, processes, prioritizes, compresses, and uploads the Float’s Status and Data Product information to the NEPI-Portal in “the Cloud.” [*This* *document* details the NEPI-Bot Application Architecture.]

## The NEPI-Portal Application

The **NEPI-Portal** Application is a wan-user’s interface, supporting the following main features: 1) Map visualization(s) of the Float’s Status and Data Products, 2) Reference dashboards of the Float’s Status and Data Products, and 3) Allow users to send commands to one or more floats (*e.g.*, new configurations). All business logic of the NEPI-Portal is handled by the NEPI API (*i.e.*, RESTful web service layer). This includes fetching data from the NEPI Storage for visualization, publishing/uploading new status and configuration information, or sending new alarms or configuration requests to the Floats. [See the “*TBD*” for detailed information regarding this Application.]

# NEPI-Bot General Architecture

## Introduction

NEPI-Bot (*see*: Figure 1 below) is a Python2-based Application comprised of several distinct subsystems and supportive class libraries, all of which are co-resident with other Float Applications and executed on the Numurus Float Device. The Float’s processor is a Zynq-7000 and is based on the Xilinx SoC architecture (dual-core or single-core ARM® Cortex™-A9 based processing system).

The NEPI-Bot Application sits strategically between 1) the Numurus SDK (*i.e.*, the Application that samples the Float’s sensors and records the acquired Status and Data Product information) and 2) various “Cloud” Applications (*i.e.*, the eventual consumers of the Float’s Status and Data Product information).

**Config**

**Files**

**Status**

**& Data**

**Files**

**Cloud**

**API**

**NEPI-Bot**

**Bot-DB**

(SQLite)

**Bot-Send**

**Bot-Recv**

**Bot-Comm**

**Other Driver**

**Wi-Fi Driver**

**RS-232 Driver**

**Ethernet Driver**

**Iridium Driver**

**NEPI**

SDK

**Float**

Figure 1: The NEPI-Bot Application with Supporting Subsystems

Using its **Bot-Recv** Subsystem, the NEPI-Bot Application processes inbound control, configuration, and alarm messages from “the Cloud” and either 1) acts directly on Bot-targeted messages or 2) passes SDK-targeted message on to the Numurus SDK for subsequent processing.

Using its **Bot-Send** Subsystem, the NEPI-Bot Application interfaces with the Numurus SDK Application in order to: 1) gather Status and Data Product records accumulated by the Numurus SDK during each “wake-up” cycle, which may be either scheduled or prompted by various alarms and triggers, 2) prioritize these records, 3) package the top-priority records, and 4) transmit them to “the Cloud.”

Using a variety of NEPI-Bot Class Libraries, all Status and Data Product records are evaluated by NEPI-Bot according to a highly-configurable priority algorithm that provides a sophisticated Priority In / Priority Out rating system (PIPO) for determining their inclusion in scheduled or triggered transmissions to “the Cloud.” Records of sufficiently-high rating are bit-compacted, compressed, signed, and, possibly, encrypted, into configurable message buffers for delivery to subscribing Applications, external to the Float itself.

Finally, the NEPI-Bot Application utilizes the Bot-Comm class library’s communications API (exposing the underlying transmission protocol) to send the contents of the message buffers “to the Cloud” for consumption by various visualization Applications.

Subsequent to this Status and Data Product consumption-prioritization-compression-transmission cycle and prior to its inevitable suspension/termination, NEPI-Bot performs various housekeeping duties to ensure data integrity and prevent an overwhelming accumulation of archived Status and Data Product records.

## NEPI-Bot Application Design

The NEPI-Bot Application is comprised of two (2) distinct, but tightly-coupled, Subsystems: Bot-Recv and Bot-Send. There are additional class and “helper” libraries, such as Bot-Defs, Bot-Cfg, Bot-DB, Bot-PIPO, Bot-Msg, Bot-Help, and Bot-Comm, that will be introduced here and described in more detail later in this document.

Both Bot-Recv and Bot-Send are awakened (“forked”) periodically by the Numurus SDK (NumSDK). These “wake-up cycles” may be scheduled or prompted by various alarms and triggers and, with each new start-up, Bot-Recv and Bot-Send each begin by accessing and consuming the latest NEPI-Bot Configuration File, a configurable, JSON-formatted file containing significant settings and attributes that drive the NEPI-Bot processing model.

Bot-Recv manages all messages inbound from “the Cloud,” which are either “Float-Alarm,” “Float-Control,” “Float-Configuration” in nature. Bot-Recv uses the Bot-Comm class library’s API to interact agnostically with the various communication protocols supported by the NEPI-Bot Application and either 1) processes Bot-targeted messages itself or 2) passes SDK-targeted messages on to the Numurus SDK.

Bot-Send interacts with the Numurus SDK to gather all Status and Data Product information, recently sampled by the Numurus SDK, for subsequent processing. The primary responsibility of Bot-Send is to: 1) prioritize both new and old (*i.e.*, previously sampled but, as yet, unsent) Status and Data Product records, 2) cleverly package them using bit-compaction and lossless compression, 3) sign them and provide a checksum if specified, 4) encrypt them when necessary, and 5) transmit them to “the Cloud.”

In addition to its major Subsystems, Bot-Recv and Bot-Send, the NEPI-Bot Application makes use of a number of Class Libraries, helper functions, and definition files, that underwrite the primary tasks of the Application. Described below, these useful objects implement Subsystem configuration, debugging, logging, database management, Status and Data Product evaluation, communications interfaces, file locking, housekeeping, and more.

## Bot-Recv Subsystem (*botrecv.py*)

Early in a Float’s wake-cycle, NEPI-Bot’s Bot-Recv Subsystem is “forked” by the Numurus SDK. The primary job of the Bot-Recv subsystem is to check for “control, configuration, and alarm” messages from “the Cloud.” These inbound (or, “downlink”) messages may be targeted for NEPI-Bot or the Numurus SDK.

In the former case, Bot-Recv decompresses, unpacks, verifies, and evaluates Bot-targeted inbound message and acts on the information immediately. In the latter case (*i.e.*, alarm, control, and configuration messages targeted for the Numurus SDK), Bot-Recv decompresses, unpacks, verifies, and creates the necessary JSON-formatted API files (per formatting described in the “NumSDK - NEPI-Bot ICD” document). After taking all necessary actions on consumed downlink messages, Bot-Recv empties the current cycle’s message queue facilities and performs other necessary housekeeping activities before suspending/terminating.

## Bot-Send Subsystem (*botsend.py*)

At an appropriate “later time” in a Float’s wake-cycle (see the “NumSDK - NEPI-Bot ICD” document), the Bot-Send Subsystem is “forked” by the Numurus SDK. In summary, Bot-Send utilizes the Bot-DB, Bot-PIPO, Bot-Msg, Bot-Help, and Bot-Comm class libraries, and other feature-rich library components to: 1) instantiate with the current configuration file settings (*see*: Bot-Cfg), 2) retrieve the latest Status and Data Product records accumulated by the Numurus SDK, 3) store these records in the Float’s embedded database (see: Bot-DB), 4) priority-evaluate each new Status and Data Product record (see Bot-PIPO), 5) re-prioritized (when needed) previously-gathered, but unsent, records, 6) create a configurable message buffer (or, buffers) for uploading to “the Cloud,” 7) bit-compact, compress, sign, and checksum, selected Status and Data Product records (*see*: Bot-Msg), 8) establish a connection with and transmit to “the Cloud” (*see*: Bot-Comm), and 9) perform necessary housekeeping prior to suspension/termination.

## Bot-Defs Definitions File (*botdefs.py*)

**Bot-Defs** is a “header-type” file used by the NEPI-Bot’s Subsystems and Class Libraries to provide essential, “hard-coded” definitions necessary to the correct functioning of those entities. Although small by design, the *botdefs.py* file contains the most axiomatic information needed to establish a common, stable, starting point from which the NEPI-Bot’s Subsystems and Class Library objects operate uniformly.

## Bot-Cfg Class Library (*botcfg.py*)

**Bot-Cfg** is a Class Library, used by the entire NEPI-bot Application to provide a configurable suite of configuration, control, and parameter settings that allow all Subsystems and Class Libraries to operate in a uniform manner. At the heart of this Class Library is the JSON-formatted “Bot Configuration File” (*config.json*) that contains dozens of important keyword-value pairs to enable the Application to function within expected boundaries. In the absence or corruption of the Bot Configuration File, Bot-Cfg generates one of three default valuations for all settings (based on internal defaults for local (unit-level) development and testing, project (integration-level) testing, or Float (production-level) deployment.

## Bot-Log Class Library (*botlog.py*)

**Bot-Log** is a Class Library, used by the entire NEPI-bot Application to provide independent debugging and logging output to the console device and/or on-board SD card, respectively. Control for debugging and logging is maintained in the Bot Configuration File and the configurable settings provide for up to twelve (12) levels (depth) of formatted detail and error message reporting.

## Bot-DB Class Library (*botdb.py*)

**Bot-DB** is a Class Library of NEPI-Bot Application, used by both the Bot-Recv and Bot-SendSubsystems to provide a DB-agnostic API for managing an embedded, self-instantiating, relational database on the Float Device. At the present time, SQLite has been selected for storage of the Status and Data Product records retrieved from the Numurus SDK application by the Bot-Send Subsystem. Bot-DB provides for database creation and instantiation, along with efficient and effective record management (*i.e.*, storage, searching, selection, sorting, field updating, primary and secondary key access, retention, and purging).

## Bot-PIPO Class Library (*botpipo.py*)

**Bot-PIPO** is a Class Library of NEPI-Bot Application, used primarily by the Bot-Send Subsystem to act as the Priority-In Priority-Out (PIPO) Manager. Bot-PIPO’s job is to take the complete list of Status and Data Product records (*i.e.*, Data Products newly-acquired from the Numurus SDK and stored in the Float’s embedded database) and compute (or, re-compute archived records when necessary) a prioritization value based on a prescribed, configurable formula. Additional Bot-PIPO class functions provide periodic low-priority record deletion, age-challenged record deletion, and general housekeeping.

## Bot-Msg Class Library (*botmess.py*)

**Bot- Msg** is a Class Library of NEPI-Bot Application, used by both the Bot-Recv and Bot-Send Subsystems to manage 1) the reception of control and configuration messages inbound from “the Cloud” and 2) the processing of request, status, and data messages outbound to “the Cloud.” Bot- Msg provides functionality to bit-pack or unpack, compress or decompress, sign or verify, checksum creation or verification, and encrypt or decrypt, depending on the inbound or outbound requirements.

## Bot-Comm Class Library (*botcomm.py*)

**Bot-Comm** is a Class Library of NEPI-Bot Application, used by both the Bot-Recv and Bot-SendSubsystems, comprised of two functional components: a convenient API and a set of supported communications drivers behind the interface. The API component is designed to be a simple, protocol-agnostic, interface that provides a uniform set of function calls to: 1) instantiate a communication object, 2) establish connections to “the Cloud,” 3) receive messages from and send messages to “the Cloud,” 4) manage protocol-related message packetization, 5) manage message acknowledgement functionality, and 6) flush and terminate unneeded connections. For purposes of this document, “the Cloud” can be described as any Application, external to the Float itself, that consumes the Status and Data Product information acquired by the Float and, subsequently, transmitted to those Applications.

## Bot-Help Library (*bothelp.py*)

**Bot-Help** is a standard function library of NEPI-Bot Application, used by both the Bot-Recv and Bot-SendSubsystems, that provides a general set of “helper” features and functionality that support Python2 enumeration techniques, configuration file consumption/updating, JSON file consumption/updating, file locking, and more.

# NEPI-Bot Application Configuration

## Introduction

When awakened by the Numurus SDK at appropriate, configurable times, all NEPI-Bot Subsystems (*i.e.*, Bot-Recv and Bot-Send) configure themselves in one of the following ways:

1. An attempt is made to retrieve the JSON-formatted, NEPI-Bot Configuration File (*i.e.*, the SD-Card resident file: *<NEPI\_home>/cfg/bot/config.json*). If retrieval of the Bot Configuration File is successful, the executing NEPI-Bot Subsystem will use the information from the Bot Configuration File to configure itself with the required configuration, control, parameter, and attribute settings. Otherwise, if retrieval of the Bot Configuration File is not successful, option #2 is taken.
2. If retrieval of the Bot Configuration File is unsuccessful, the NEPI-Bot Subsystems have three (3) sets of built-in, default values for all configuration, control, and parameter settings. These defaults are based on development-sensitive (unit-level) testing, Numurus-sensitive (end-to-end, or integration-level) testing, and Float-sensitive (production, or “in-water”) deployment. In addition to using these hard-wired, limited-functionality, “final option” settings to configure itself, the executing NEPI-Bot Subsystem will use these values to attempt creating of a valid Bot Configuration File on the SD Card (available, if successful, for subsequent executions of any NEPI-Bot Subsystem).

For details regarding the format, contents, defaults, and uses of the NEPI-Bot Application’s configuration, control, parameter, and attribute settings, see the “The Bot Configuration File (*config.json*)” below.

## The NEPI-Bot Configuration File (*config.json*)

The NEPI-Bot Configuration File provides configuration, control, parameter, and attribute settings to the various NEPI-Bot Application Subsystems and Class Library components. The name of this configuration file is *config.json*, its format is JSON, and its location is in the <*NEPI\_home*>/cfg/bot/ directory. The contents of the NEPI-Bot Configuration File are described in Table 1 below.

In order to instantiate themselves fully, all Subsystems and Class Library components comprising the NEPI-Bot Application assume the existence of the NEPI-Bot Configuration File. Each Subsystem begins by instantiating a Bot-Cfg Class Object early in the execution. The Bot Configuration File itself is consumed once and its control, configuration, attribute, and parameter settings are stored in Class Variables of the Bot-Cfg Object to be used by the Subsystem directly or passed as a Class Object to other Class Libraries during the Subsystem’s execution.

In the absence of this critical configuration file, a set of built-in Factory Default Values are implemented for all control, configuration, attribute, and parameter settings to enable the executing NEPI-Bot Subsystem to function properly, albeit in a limited or degraded capacity. However, it should be noted that production-level reliance on the efficacy of the Default Configuration Values involves certain risks. Bot Config File contents are described in the Table below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Element Name** | **JSON Type** | **Range/Format** | **Description/Notes** |
| *state* | string | **[ ut | it | fl ]**  ut Factory Default: ***ut***  it Factory Default: ***ut***  fl Factory Default: ***fl*** | **Running State**  Platform the application is running on; ut = Unit Test, it = Integration Test, and fl = Float (production); this value drives some internal settings and code logic. |
| *debugging* | integer | **[ -1, 0 - 23 ]**  ut Factory Default: ***11***  it Factory Default: ***6***  fl Factory Default: ***-1*** | **Debug Mode**  (used to control execution of internal coding and debug output to console; ‘-1’ is OFF; the higher the debug level, the more detailed the report and the more indentation in output; levels 12-13 revert to levels 0-11 for indentation; 2 spaces indent for each level) |
| *logging* | integer | **[ -1, 0 - 11 ]**  ut Factory Default: ***11***  it Factory Default: ***6***  fl Factory Default: ***-1*** | **Logging Mode**  (used to control execution of internal coding and debug output to console; ‘-1’ is OFF; the higher the logging level, the more detailed the report and the more indentation in output; levels 12-23 revert to levels 0-11 for indentation; 2 spaces indent for each level) |
| *timing* | boolean | **[ 0 | 1 ]**  ut Factory Default: ***1***  it Factory Default: ***1***  fl Factory Default: ***1*** | **Time Flag**  (0=False, 1=True; determines whether the current ‘time’ is prepended to each output line in both debugging and logging) |
| *locking* | integer | **[ 0 | 1 ]**  ut Factory Default: ***0***  it Factory Default: ***0***  fl Factory Default: ***0*** | **Locking Active Flag**  (0=False, 1=True; determines whether file locking is used during file access in Bot Subsystem execution) |
| *comms* | integer | **[ 0 | 1 ]**  ut Factory Default: ***0***  it Factory Default: ***1***  fl Factory Default: ***1*** | **Communications Active Flag**  (determines whether or not the Subsystem will instantiate and use the specified communications protocol 0 = no comms will be attempted; 1 = comms will be used. Turning off the comms allows s Subsystem to run through without failing due to a lack of communications; good for testing purposes) |
| *type* | string | **[ Iridium | Ethernet | Wi-Fi ]**  ut Factory Default: “***Wi-Fi***”  it Factory Default: ***“Ethernet”***  fl Factory Default: ***“Iridium”*** | **Communications Type**  (describes the primary communications protocol in sending information to the Cloud; three are available through 7/1; more to follow in commercial development) |
| *host* | string | **Class A, B, or C IP Addresses**  ut Factory Default: ***10.0.0.116***  it Factory Default: ***18.223.151.16***  fl Factory Default: ***N/A*** | **Host IP Address**  (the IP address used by the NEPI-Bot Application’s Bot-Comm Class Library for socket connections) |
| *port* | integer | **[2048 - 65535]**  ut Factory Default: ***COM4*** it Factory Default: ***/dev/ttyUSB0***  fl Factory Default: ***/dev/ttyUSB0*** | **Port**  (along with “Host IP Address” above, the ‘port’ completes the destination or origination network address of a Float-to-Cloud or Cloud-to-Float message; may not be used in the case of “Iridium” comms) |
| *baud* | integer | **[ 110 - 921600 ]**  ut Factory Default: ***19200***  it Factory Default: ***19200***  fl Factory Default: ***19200*** | **Communication Baud Rate**  (although any common bits per second are acceptable, the most common are 1200, 2400, 4800, 9600, 19200, 38400, 57600, and, 115200; lower and higher are supported if the non-float device supports them) |
| *tout* | integer | **[ 0 - n ]**  ut Factory Default: ***3***  it Factory Default: ***3***  fl Factory Default: ***3*** | **Serial Port Timeout**  (integer value; used in instantiating the serial port using the serial.Serial() method) |
| *isp\_open\_attm* | integer | **[ 1 - n ]**  ut Factory Default: ***10***  it Factory Default: ***10***  fl Factory Default: ***109*** | **Serial Port Open Attempts**  (integer value; the number of times an open() of the serial port will be attempted)) |
| *isp\_open\_tout* | integer | **[ 1 - n ]**  ut Factory Default: ***1***  it Factory Default: ***1***  fl Factory Default: ***1*** | **Serial Port Open Time Delay**  (integer value; number of seconds delay between serial port open attempts) |
| *protocol* | integer | **[ 0 - 4 ]**  ut Factory Default: ***2***  it Factory Default: ***2***  fl Factory Default: ***1*** | **Communications Protocol**  (defines the communications method being used by the Float, where ‘0’ represents “OFF,” 1= Iridium, 2=Ethernet, 3=RS-232, and 4=Wi-Fi; has become obsolete; will be revived in commercial development) |
| *packet\_size* | integer | **[ n ]**  ut Factory Default: ***1024***  it Factory Default: ***1024***  fl Factory Default: ***340*** | **Protocol Packet Size**  (used in conjunction with ‘protocol,’ Packet Size represents the desired or required packet size for the selected protocol; should be optimized for protocol) |
| *sys\_status\_file* | string | **<*file\_name*>.json**  ut Factory Default: ***sys\_status.json***  it Factory Default: ***sys\_status.json***  fl Factory Default: ***sys\_status.json*** | **Status File Name**  (contains date-specific system state and status information to be uploaded to the Cloud; see SDK-Bot ICD for detailed information) |
| *data\_dir* | string | **<*relative\_path*>**  ut Factory Default: ***data***  it Factory Default: ***data***  fl Factory Default: ***data*** | **Data Directory**  (location of the Float’s Data Directory; relative to the Float’s home directory; avoid leading and trailing “/” chars) |
| *data\_zlib* | integer | **[ 0 | 1 ]**  ut Factory Default: ***1***  it Factory Default: ***1***  fl Factory Default: ***1*** | **Zlib Flag**  (0=False, 1=True; flag indicates whether or not the zlib() library method is used to decompress the downlink message data component) |
| *data\_msgpack* | integer | **[ 0 | 1 ]**  ut Factory Default: ***1***  it Factory Default: ***1***  fl Factory Default: ***1*** | **Msgpack Flag**  (0=False, 1=True; flag indicates whether or not the msgpack() library method is used to unpack the downlink message data component) |
| *log\_dir* | string | **<*relative\_path*>**  ut Factory Default: ***log***  it Factory Default: ***log***  fl Factory Default: ***log*** | **Log Directory**  (location of the Float’s Log Directory, relative to the Float’s home directory; avoid leading and trailing “/” chars) |
| *br\_log\_name* | string | **<*file\_name*>.txt**  ut Factory Default: ***brlog.txt***  it Factory Default: ***brlog.txt***  fl Factory Default: ***brlog.txt*** | **Bot-Recv Log File Name**  (contains the Bot-Recv logging information output by the Bot-Log Class Library defined in *botlog.py*) |
| *bs\_log\_name* | string | **<*file\_name*>.txt**  ut Factory Default: ***bslog.txt***  it Factory Default: ***bslog.txt***  fl Factory Default: ***bslog.txt*** | **Bot-Send Log File Name**  (contains the Bot-Send logging information output by the Bot-Log Class Library defined in *botlog.py*) |
| *bu\_log\_name* | string | **<*file\_name*>.txt**  ut Factory Default: ***bulog.txt***  it Factory Default: ***bulog.txt***  fl Factory Default: ***bulog.txt*** | ***Unknown* Log File Name**  (contains logging information for an indeterminate Subsystem execution as output by the Bot-Log Class Library defined in *botlog.py*) |
| *wt\_changed* | integer | **[ 0 | 0 ]**  ut Factory Default: ***0***  it Factory Default: ***0***  fl Factory Default: ***0*** | **Weight Factor Changed**  (specifies whether any numerator weight factor has changed since the last wake-up due to a Config & Control Message from “the Cloud,” thus necessitating a PIPO Rating re-calculation for all DB-archived Data Products) |
| *pipo\_scor\_wt* | float | **[ 0.0 - 1.0 ]**  ut Factory Default: ***0.5***  it Factory Default: ***0.5***  fl Factory Default: ***0.5*** | **Score Weight Factor**  **Quality Weight Factor**  **Size Weight Factor**  **Time Weight Factor**  **Trigger Weight Factor**  (each of these values are cloud-configurable “weights” used in the PIPO prioritization formula:  ((pipo\_scor\_wt \* Score) + (pipo\_qual\_wt \* Quality) + (pipo\_size\_wt \* Size) + (pipo\_trig\_wt \* Trigger)) / (pipo\_time\_wt \* Time) + 1.0 where the prioritization element values are derived from metadata stored along with Data Products during a Float’s wake cycle (see the Data Product Prioritization (PIPO Manager: *botpipo.py*) section of this document) |
| *pipo\_qual\_wt* | float | **[ 0.0 - 1.0 ]**  ut Factory Default: ***0.5***  it Factory Default: ***0.5***  fl Factory Default: ***0.5*** |
| *pipo\_size\_wt* | float | **[ 0.0 - 1.0 ]**  ut Factory Default: ***0.5***  it Factory Default: ***0.5***  fl Factory Default: ***0.5*** |
| *pipo\_trig\_wt* | float | **[ 0.0 - 1.0 ]**  ut Factory Default: ***0.5***  it Factory Default: ***0.5***  fl Factory Default: ***0.5*** |
| *pipo\_time\_wt* | float | **[ 0.0 - 1.0 ]**  ut Factory Default: ***1.0***  it Factory Default: ***1.0***  fl Factory Default: ***1.0*** |
| *purge\_rating* | float | **[ 0.0100 - 0.9000 ]**  ut Factory Default: ***0.0500***  it Factory Default: ***0.0500***  fl Factory Default: ***0.0500*** | **Purge Rating**  (minimum PIPO Rating, below which the Data Product will be purged (deleted) from the Float’s embedded database) |
| *max\_msg\_size* | integer | **[ 1024 - n ]**  ut Factory Default: ***1.0 MB***  it Factory Default: ***680***  fl Factory Default: ***340*** | **Maximum Message Size**  (max byte size of the outbound Cloud msg after bit-packing, signing, compression, checksum, encryption, and/or etc.) |

Table 1: The NEPI-Bot Configuration File (JSON Formatted)

## Internal Defaults

The following is a sample JSON-formatted Bot Configuration File (in use during the time of this document version), along with the Internal Default Values for Unit Testing (ut), Integration Testing (it), and Production Deployment (fl):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Unit Test DEFAULT** | **Integration DEFAULT** | **Float DEFAULT** | | ut | it | fl | | 11 | 11 | -1 | | 11 | 11 | -1 | | 1 | 1 | 1 | | 0 | 0 | 0 | | 1 | 1 | 1 | | Wi-Fi | Ethernet | Iridium | | 10.0.0.116 | 18.223.151.16 | 52.38.4.219 | | COM4 | /tty/USB0 | /tty/USB0 | | 19200 | 19200 | 19200 | | 3 | 3 | 3 | | 5 | 5 | 10 | | 1 | 1 | 1 | | 2 | 2 | 1 | | 2048 | 1024 | 370 | | sys\_status.json | sys\_status.json | sys\_status.json | | data | data | data | | 1 | 1 | 1 | | 1 | 1 | 1 | | log | log | log | | brlog.txt | brlog.txt | brlog.txt | | bslog.txt | bslog.txt | bslog.txt | | bulog.txt | bulog.txt | bulog.txt | | 0 | 0 | 0 | | 0.5 | 0.5 | 0.5 | | 0.5 | 0.5 | 0.5 | | 0.5 | 0.5 | 0.5 | | 0.5 | 0.5 | 0.5 | | 1.0 | 1.0 | 1.0 | | 0.05 | 0.05 | 0.05 | | 1024 | 680 | 340 | | **ALPHA DEFAULT** | **TEST DEFAULT** | **FLOAT DEFAULT** |   {  "state": ut,  "debugging": 3,  "logging": 5,  "timing": 1,  "locking": 0,  “comms”: 1,  “type”: “Iridium”,  "host": "10.0.0.116",  "port": 7770,  “baud”: 19200,  “tout”: 3,  “isp\_open\_attm”: 10,  “isp\_open\_tout”: 1.  "protocol": 2,  "packet\_size": 1024,  "sys\_status\_file": "sys\_status.json",  "data\_dir": "data",  “data\_zlib”: 1,  “data\_msgpack”: 1,  "log\_dir": "log",  "br\_log\_name": "brlog.txt",  "bs\_log\_name": "bslog.txt",  "bu\_log\_name": "bulog.txt",  "wt\_changed": 0,  "pipo\_scor\_wt": 0.5,  "pipo\_qual\_wt": 0.5,  "pipo\_size\_wt": 0.5,  "pipo\_trig\_wt": 0.5,  "pipo\_time\_wt": 1.0,  "purge\_rating": 0.05,  "max\_msg\_size": 1000000  } |

Figure 2: Sample JSON-formatted Bot Configuration File (config.json) with Defaults

# NEPI-Bot Debugging/Logging (the *botlog.py* Class Library)

## Introduction

Debugging and logging methodologies have been provided for the NEPI-Bot Subsystems, along with their supporting Class and helper-function libraries, which allow for “live” (*i.e.*, console) and “permanent” (*i.e.*, file storage) tracking of milestones, events, and errors.

Both tracking modes - Debugging and Logging - display information as formatted, structured output in a line-by-line style with optional date timestamps. Each is activated independently with “debugging” intended for use primarily during unit/integration-level development and testing and “logging” intended primarily for FLOAT (production-level) deployment *when the need arises* - although, arguably, file “logging” can be used effectively during any project phase.

Debugging and Logging is implemented by the BotLog Class Library (*botlog.py*) and, although they share the same instantiation and subsequent method calls, each of these ‘tracking’ components operates independently.

## Debugging Mode

Debugging Mode is defined as formatted, structured output to “standard out” (typically, the “console”) during development on platforms that support console output - or, at least, the ability to redirect it appropriately. Usually, there is no reason to redirect Debugging’s standard output to a file since that’s the reasoning behind the “logging” capability (see below).

## Logging Mode

Logging is defined as formatted, structured output to a file, typically located on a platform’s storage device during development or the SD-card for Float deployment. A separate log file is created for each of the various Bot Subsystems according to its name as defined in the Bot Configuration File (*cfg.br\_log\_name* for the Bot-Recv Log File and *cfg.bs\_log\_name* for the Bot-Send Log File). If a specific Bot Log File cannot be determined, an “unknown” Log File is created (*cfg.bu\_log\_name*).

## Class Instantiation

A Bot-Log Class Library object is instantiated once during the earliest stages of a Bot Subsystem execution. The example below is taken from botsend.py and immediately follows the instantiation of the Bot-Cfg Class Library object instantiation.

|  |
| --- |
| **Code Snippet:**  . . .  ########################################################################  ## Instantiate Bot-Send Debug/Log Object (from 'botlog.py').  ########################################################################  log = BotLog(cfg, “Bot-Send”)  log.initlog(0)  . . . |

Figure : Sample Code Depicting Instantiation of BotLog Class Object

The *BotLog* constructor expects the *BotCfg* Class object (‘cfg’) - created earlier - as the first argument. Among other things, the ‘cfg’ object contains the *cfg*.*tracking*, *cfg*.*debugging*, *cfg*.*logging*, and *cfg*.*timing* Flags which are significant to the Debugging and Logging operations.

The second constructor argument (type=‘Bot-Send) indicates that this is a Log File for the Bot-Send Subsystem. The other recognized “type” is Bot-Recv if this were intended as a Log File for the Bot-Recv Subsystem. This argument is used to generate appropriate titles, names, and log files for each of the respective Subsystems. If additional Subsystems are created in the future, the context of this argument may change.

## Class Initialization

After Class Instantiation, an immediate call is made to *log.initLog()* for the purpose of initializing the Debugging and/or Logging process. The single argument (level=‘0’) simultaneously represents 1) the maximum Debugging and/or Logging setting at which any tracking information generated by the BotLog initializer will be output and 2) the starting point for expected indentation. In this case, because the initializer is being called early in the execution of the Subsystem, Debugging and/or Logging output should still be left justified as much as possible (*i.e.*, starting at the 0th column of text). Output originating from a Subsystem’s inline code will typically start out left justified with subsequent calls to various Class and Helper Library Methods progressively indented to create visually organized messages.

If neither Debugging nor Logging has been indicated (*i.e.*, neither *cfg.debugging* nor *cfg.logging* is True, respectively), nothing happens in this Method. If Debugging is indicated, Debugging starts off a fresh “heading,” output to the console. If Logging is indicated, the call to this Method either 1) creates the Log File for the first time or 2) “clears” an existing Log File of the same name - in either case, the file is loaded with a fresh “header.”

|  |
| --- |
| **Debugging Output to Console:**  Wed Mar 8 17:42:18 2019: STARTING BOT-SEND DEBUGGING:  Wed Mar 8 17:42:18 2019:  . . .  **Logging Output to Log File:**  Wed Mar 8 17:42:18 2019: STARTING BOT-SEND LOGGING:  Wed Mar 8 17:42:18 2019:  . . . |

Figure : Output Snippet Showing Initialization of 1) Console Debugging and 2) File Logging

If Debugging and/or Logging has been indicated, tracking information regarding the earlier instantiation of the Bot Configuration object, including information from its initialization process, is appended to the Debugging and/or Logging output.

## Configuration Parameters

Found in the Bot Configuration file, both Debugging (*cfg.debugging*) and Logging (*cfg.logging*) offer twenty-four (24) levels of detail, represented by integer value settings which range [ -1, 0-11, 12-23 ]. These “levels” simultaneously determine 1) the depth of detail for which output will be generated and 2) the amount of indentation prepended to each line of output (which, of course, contributes significantly to the form and readability of the output).

A Debugging or Logging value of -1 indicates that the functionality is NOT in use (*i.e.*, OFF) during this execution of the Subsystem (along with its imported Class and/or Helper Libraries). Subject to further discussion below, any setting *0-23* indicates that the respective Debugging or Logging functionality is in use (*i.e.*, ON).

A Debugging or Logging value of 0-11 indicates output at an ever-increasing level of detail. The output for each level of detail is indented a number of spaces equivalent to twice the indicated level. For example, a debug/log level of 4 would yield an indent of eight (8) spaces prepended to the anticipated output text.

A Debugging or Logging value of 12-23 also indicates output at an ever-increasing level of detail. However, indentation reverts back to the debug/log levels of 0-11, respectively. That is, a debug/log level or 14 represents a fairly low level of detail, but with an indentation equivalent of a debug/log level of 2. This paradigm allows for display of certain related information to be output only at much higher levels of debugging and/or logging.

For programming convenience, a useful configuration flag (*cfg.tracking*) has been implemented that is True if either *cfg.debugging* or *cfg.logging* is ON (*i.e.*, set to *0-23*). If both Debugging and Logging are set to *-1* in the Bot Configuration file, *cfg.tracking* is set to False. The purpose of this flag allows programmers to restrict making calls to the *BotLog* Class Library Methods only when Debugging and/or Logging has been indicated (without having to test for each one separately).

If the Timing Flag (*cfg.timing*) is set to *1* in the Bot Configuration file (*i.e.*, True), then a date timestamp is prepended to each line of Debugging or Logging output. Otherwise, only the Debugging or Logging text is output (both subject to appropriate indentation, specified by the specified debugging/logging “levels”). The code snippet and output in Figure 4 below demonstrates how both Debugging and Logging output is generated.

|  |
| --- |
| **Code Snippet:**  . . .  ########################################################################  # Re-Evaluate PIPO Ratings for Archived Data Products if Required.  ########################################################################  if cfg.tracking:  log.track(0, "Recalculate Archived PIPO Ratings.", True)  log.track(1, "Select 'Active' Records from Database.", True)  for row in rows:  if cfg.tracking:  log.track(1, "Re-evaluating Data Product ID: " + str(rwid), True)  log.track(2, "Numr: " + str(numr), True)  log.track(2, "Trig: " + str(trig), True)  log.track(2, "Qual: " + str(qual), True)  . . .  **Output:**  . . .  Wed Mar 8 17:42:18 2019: Recalculate Archived PIPO Ratings.  Wed Mar 8 17:42:18 2019: Select 'Active' Records from Database.  Wed Mar 8 17:42:18 2019: Reevaluating Data Product ID: 5  Wed Mar 8 17:42:18 2019: Numr: 1.346265  Wed Mar 8 17:42:19 2019: Trig: 0.5  Wed Mar 8 17:42:19 2019: Qual: 0.7575  . . . |

Figure : Sample Code Snippet with Anticipated Debugging/Logging Output

## Primary Output Method (*log.track()*)

Calls to the BotLog Class Library’s primary output Method (*log.track()*) are executed if and only if *cfg.tracking* is True (*i.e.*, Debugging and/or Logging has been indicated). The *log.track(lev, msg, new)* Method requires a “level,” representing 1) the maximum Debugging and/or Logging setting at which this message will be output and 2) the required indentation. The ‘msg’ argument is a string value that will be output “as is.” The final argument (‘new’) is a True/False value indicating whether a “newline” is to be appended to the output message. This feature allows for additional processing to take place between, for example, the output of a “keyword” or “label” followed by the output of a calculated “value” shortly thereafter.

For a “level” of 0-11, Debugging or Logging is ON and, if a call to the *log.track()* Method is made, indentation equal to two (2) spaces for each “level” is prepended to the message. The means, for example, that a “level” of 0 is fully left-justified (0 spaces) and a level of 4 is indented twelve (8) spaces.

For a “level” of 12-23, Debugging and/or Logging is also ON and these higher “levels” indicate even more detailed output. However, the indentation for “levels” 12-23 equates to levels 0-11, respectively. That is, “level” six (12) will only be output if Debugging and/or Logging is set to 12 or above, but the indentation will be zero (0) spaces (*i.e.*, indentation equivalent to level=0).

|  |
| --- |
| **Code Snippet:**  . . .  if cfg.tracking:  log.track(4, "Acquired: " + str(acquired), True)  log.track(16, "Contents: " + str(metajson), True)  . . .  **Output:**  . . .  Wed Mar 8 17:42:19 2019 Acquired: True  Wed Mar 8 17:42:19 2019 Contents: { json file contents }  . . . |

Figure : Sample Code Snippet Demonstrating Debugging/Logging Level 9

In Figure 6 above, the intent is to output the label of “Acquired” and the value of “True” only when Debugging and/or Logging Mode is set to “4” or above. This also means that, if this message is output, an indentation of eight (8) spaces is desired. On the other hand, we only want to see the “Contents” of the JSON file that was acquired if we increase our desire for detail up to “14” or above. The reason we chose “14” is that the indentation of the “Contents:” line will match the indentation of the line above it (*i.e.*, the “Acquired:” output line).

## Error Output Method (*log.errtrack()*)

The ***log.errtrack()*** Method is still in use - primarily to support the existing code base - but has been deprecated in the **bot61** Bitbucket branch release. The Method will be reinstituted in subsequent commercial branch releases (anticipated: **cbot71**) that will sport an extended argument list and additional feature capabilities.

# Communications Manager (*botcomm.py*)

## Introduction

Bot-Comm is a Class Library of the NEPI-Bot Application, used by the Bot-Sendsubsystems, and is comprised of two functional components: a convenient API and a set of supported communications drivers behind the interface.

The API component is designed to be a protocol-agnostic interface that provides a uniform set of function calls, buffering all NEPI-Bot subsystems from the intricacies of communications handshaking, packet sizing, packet ordering and reconstruction, message acknowledgements, error correction, etc. The API provides the Application with a simple way to: 1) establish connections to “the Cloud,” 2) receive messages from and send messages to “the Cloud,” 3) manage protocol-related message packetization, 4) manage message acknowledgement functionality, and 5) flush and terminate unneeded connections.

The primary *(i.e.*, production default) transmission protocol is Iridium, using the Iridium Short Burst Device on the Float and communicating with the Iridium Subscriber Unit in “the Cloud.” Other communication protocols, like Ethernet, RS-232, Wi-Fi, etc., provide alternative delivery methods and useful testing capabilities.

## Communication Object Instantiation

The Bot-Comm Class Library’s API provides for the creation of a new communication object in the following manner:

from botcomm import BotComm

...

bc = BotComm(cfg\_object *cfg*, log\_object *log*, str *comm\_type*, int *log\_level*)

where the ‘*cfg*’ and ‘*log*’ objects are those instantiated earlier in a subsystem’s execution, the ‘*comm\_type*’ is one of the authorized communication methods (*e.g.*, ”Iridium,” “Ethernet,” “Wi-Fi,” etc.), and the ‘*log\_level*’ represents the debugging and logging depth (and indentation) desired.

## Communications Connection

Once a communications object is created, an actual connection can be established anytime thereafter as follows:

success = bc.getconn(int *log\_level*)

Depending on the communication protocol selected, it is possible that the actual communications connection might be established differently. For example, one protocol might establish its connection in the class constructor with the subsequent call to *getconn()* doing little or nothing. Using another protocol, the class constructor might simply return the class object with the establishment of the connection occurring later during the call to *getconn()*. In all cases, however, communications should be established uniformly by instantiating the communications object first, followed by a call to the *getconn()*Method.

## Downlink Message Processing

## Uplink Message Processing

## Message Acknowledgement

*in progress with Kevin/Jacob …*

## Communications Termination

Because communication connections may be handled differently depending on the protocol (e.g., socket connections in Python typically employ shutdown() followed by close()), Bot-Comm implements a by-the-protocol approach to connection closure. The API provides a single Class Method call to insure all communications are ended in a safe and secure manner.

bc.close()

The call to *close()*destroys the Class object, thus requiring a new class instance in the case where subsequent communications operations are envisioned.

# Bot-Recv Subsystem (the NEPI-Bot ‘Downlink’ Manager: *botrecv.py*)

## Introduction

Along with Bot-Send, **Bot-Recv** is one of two primary Subsystems in the NEPI-Bot Application and it is the least complex of the two. Bot-Recv is awakened (“forked”) by the Numurus SDK according to a configurable, periodic schedule. Typically, this awakening occurs once the SDK has completed sampling and saving Status and Data Product information in one or more, date-named Data Folders located in the Float’s official “Data Directory.”

Bot-Recv utilizes feature-rich library components (*e.g.*, Bot-Defs, Bot-Cfg, Bot-Log, Bot-Help, Bot-DB, and Bot-Comm) to accomplish its primary tasks, which include:

1. *Subsystem Instantiation* by consuming The NEPI-Bot Configuration File (JSON Formatted) (as described above) and using the Bot-Cfg Class Library functionality to implement the various configuration, control, parameter, and attribute settings stored in that file in prepare for execution.
2. *Retrieving ‘Downlink’ Configuration and Control Messages* by using the Bot-Comm Class Library to establish a connection with “the outside world” (*i.e.*, “the Cloud”), consuming all pending Downlink Messages from “the Cloud,” managing acknowledgements, and terminating the connection.
3. *Processing Each Downlink Message* by parsing through each Downlink Message in search of one or more embedded, Float-bound Configuration and Control Messages, at which point Bot-Recv will either 1) act directly on the Configuration and Control Messages for which NEPI-Bot itself is responsible and/or 2) pass along any SDK-related Configuration and Control messages using one or more related “SDK notification files.”
4. *Executing General Housekeeping Activities* by employing various functions in the Bot-Help and other Class Libraries to perform all necessary clean-up requirements.

## Assumptions

There are a number of assumptions (or, “givens”) upon which the Bot-Recv Subsystem relies in order to be successful; these assumptions are as follows:

1. Bot-Recv must be awakened by the Numurus SDK at appropriate time intervals. Based on other Float-related and Program-level documents, Bot-Recv is called a minimum of “8 times a day at 3 hr. intervals” (all of which are adjustable based on current Float conditions).
2. When using Iridium, Bot-Recv should be allowed a 10-minute window to account for Iridium connection behavior (based on observation and testing extensive ‘Check Signal Strength’ responses).

*Additional Assumptions TBD …*

## Subsystem Configuration

When awakened by the Numurus SDK at appropriate, configurable times, Bot-Recv configures itself by accessing the necessary settings from the Bot Configuration File using the Bot-Cfg Class Library (***botcfg.py***) or, as a last resort, loading internal, built-in default values. For details regarding the format, contents, defaults, and uses of the configuration, parameter, and attribute settings, see the “NEPI-Bot Subsystem Configuration” section above.

## Control and Configuration Message Retrieval

Subsequent to instantiation, Bot-Recv attempts to retrieve up to fifty (50) so-called “downlink” messages from the Cloud (the maximum number of queued Iridium MT messages). To do so, Bot-Recv invokes Bot-Comms receive() Class Library Method as follows:

success, cnc\_msgs = bc.receive(int *log\_level*, int *max\_msgs*)

where log\_level is the desired debugging/logging level (with its associated indentation) and max\_msgs represents the maximum number of messages desired (in the case of “Iridium,” this translates to five, 270-byte, Iridium MT messages, sufficient to transmit approximately 50 embedded control and configuration segments).

## Control and Configuration Message Processing

If communication messages are received from the Cloud, each message is deconstructed segment by segment, where each segment is a separate and distinct Float-targeted control or configuration directive. Configuration and control “segments” are described in detail in the NEPI-API ICD and the general format is reprised here for convenience:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Configuration Message Map** | | **Element Name** | **Range/Format** | **Description/Notes** |
| **Bytes 1-2 (16-bit integer)** | **0-1**  (2 bits) | *protocol* | [0-3] | **Protocol**  (defines the version of this specification; this version is 0; this gives us room to iterate on the protocol 3 more times) |
| **2-4**  (3 bits) | *config\_type* | [ 0 - 7 ] | **Configuration Type**  (defines the type of configuration. 0=command, 1=sensor, 2=node, 3=geofence, 4=rule, 5=trigger, 6=action, 7=schedule) |
| **5-15**  (11 bits) | *message\_length* | [ 0 - 2048] | **Message Length**  (the number of bytes in the actual control and configuration ‘message,’ that is, the segment’s payload in bytes 4-n) |
| **Byte 3 (8-bit integer)** | **16-21**  (6 bits) | *config\_index* | [ 0 - inf ]  ***mod 64*** | **Configuration Index**  (this is the new index value for the configuration type; this index value should not already exist on the float; rollover will need to be anticipated by the float’ these bits should be ignored for command configuration types) |
| **22-23**  (2 bit) | *message\_flags* | [ 0 | 1 ] | **Message Flags**  (“01” = Bot Config File changes; primarily PIPO-related; 00, 10, and 11, are reserved) |
| **Bytes**  **4-n** | **24-n** | *message* | All data of length message\_length described above. | **Message**  (the message and any message type specific headers ) |

Table : General Bit-Mapped Format of the Downlink Message

Control segments are directives that instruct NEPI-Bot and/or the SDK to take specific and, in most cases, immediate action(s). For example, a ‘scuttle’ command is a top-priority message passed to the SDK by NEPI-Bot, requiring the SDK to sink the Float at the next available opportunity (*i.e.*, immediately).

Configuration segments are directives that instruct NEPI-Bot and/or the SDK to “re-configure” themselves according to the updated parameters contained in the segment’s “message” - a compressed and packed JSON file of keyword-value pairs - that is recovered and placed in the official ‘cfg’ directory, from where NEPI-Bot and/or the SDK will take on new operational characteristics in a timely fashion.

Each of the configuration payloads has been packed and compressed using the msgpack and zlib libraries, respectively. On the Float, these payloads are unpacked and decompressed to yield the actual JSON-formatted configuration file which is, then, placed in its corresponding configuration directory according to the Table (reprised from the Numurus SDK - NEPI-Bot ICD):

|  |  |  |  |
| --- | --- | --- | --- |
| **Interface Element** | **Root Folder  (in /home/nepi-usr/)** | **Sub-directory** | **Filename(s)** |
| Control | commands/ |  | scuttle |
| Configuration | cfg/ | bot/ | config.json |
| action/ | action\_seq\_<INDEX>.json |
| sched/ | task\_< INDEX>. json |
| trig/ | smarttrig\_cfg\_<INDEX>.json |
| sensors/ | sensor\_cfg\_<INDEX>.json |
| rules/ | smarttrig\_rule\_<INDEX>.json |
| proc\_nodes/ | proc\_node\_cfg\_<INDEX>.json |
| geofence/ | geofence\_cfg\_<INDEX>.json |

Table : Interface File List (Reprised from the NumurusSDK - NEPI-Bot ICD).

At the time of this writing, the only Command recognized by NEPI-Bot (and, subsequently, the Numurus SDK) is the so-called ‘scuttle’ Command. This Command is passed by NEPI-Bot to the SDK via the ‘commands’ directory (located in *<NEPI\_home>*. A zero-length file, called scuttle, is placed there by NEPI-Bot, followed by an SDK Notification (*i.e.*, the ‘forking’ of a process to execute the notification shell script, /opt/numurus/ros/nepi-utilities/process-updates.sh).

## Subsystem Housekeeping

Upon termination - and to relieve some of the processing burden off the more-complex Bot-Send Subsystem - Bot-Recv performs extensive clean-up on the Float’s embedded database.

1. Status Records that have been ‘sent’ to the Cloud are purged from the database.
2. Data Products that have been ‘sent’ to the Cloud are purged from the database.
3. Status Records greater than ten (10) days old are purged from the database.
4. Data Products greater that ten (10) days old are purged from the database.
5. Data Products with a PIPO Rating less than the configurable Purge-Rating are purged from the database

# Bot-Send Subsystem (the NEPI-Bot Uplink Manager: *botsend.py*)

## Introduction

Along with Bot-Recv, **Bot-Send** is one of two primary Subsystems in the NEPI-Bot System and it is certainly the more complex of the two. Bot-Send is awakened (“forked”) by the Numurus SDK once the SDK has completed sampling and saving all Status and Data Product information in date-named folders in the Float’s official “Data Directory.”

Bot-Send utilizes feature-rich library components (*e.g.*, Bot-Defs, Bot-Cfg, Bot-Log, Bot-DB, Bot-PIPO, Bot-Msg, Bot-Help, and Bot-Comm) to accomplish its primary tasks, which include:

1. *Subsystem Instantiation* by consuming the JSON-formatted Bot Configuration File (as described earlier) and using the Bot-Cfg Class Library functionality to implement the various configuration, control, parameter, and attribute settings in that file to prepare for execution.
2. *Status and Data Product Retrieval* by compiling a list of all folders in the Float’s official “Data Directory” (per specifications in the “NumSDK - NEPI-Bot ICD” document) and, using the Bot-DB Class Library functionality, storing the contents of each of those folders (*i.e.*, one Status file and *n*-number of Data Product files) in NEPI-Bot’s embedded, relational database.
3. *Prioritizing Data Products* by using the Bot-PIPO Class Library component and other Class Library functionality to evaluate each new Data Product record according to a prescribed formula (see: Data Product Prioritization, elsewhere in this document) and, when necessary, re-evaluating previously-prioritized Data Products (as retained in the Float’s embedded database) that were unable to be sent in previous transmissions to “the Cloud.”
4. *Creating a Message* for eventual transmission to “the Cloud” by using Bot-Msg, Bot-Help, and other Class Library functionality to select the highest-priority Data Products that will fit into a configurable message buffer and, then, bit-compacting, compressing, signing, check-summing, and, when required, encrypting, those Status and Data Products records to maximize the value of the transmission (as limited by both size and wake-cycle period).
5. *Transmitting the Message* by using the Bot-Comm Class Library to establish a connection with “the outside world” (*i.e.*, “the Cloud”), transmitting the entire message buffer to “the Cloud,” managing acknowledgements, and terminating the connection.
6. *Executing Housekeeping Activities* by employing various functions in the Bot-Help and other Class Libraries to identify successfully-transmitted Status and Data Product records, remove them from the Float database, remove all associated folders from the official “Data Directory” (as described in the “NumSDK - NEPI-Bot ICD”), and executing other necessary clean-up requirements.

## Assumptions

There are a number of assumptions (or, “givens”) upon which the Bot-Send Subsystem relies in order to be successful; these assumptions are as follows:

1. is awakened by the Numurus SDK at appropriate time intervals. While ultimately configurable, the 12/01/2018 “Float and Data Attributes Report – Concept Phase” (Seawall) says the following about wake-up dealing with Data Product transmission to “the Cloud:” “… uploads are performed 8 times a day at 3 hr. intervals.”
2. assumes the existence of the Float’s official Data Directory: *<NEPI\_home/data*. No provision has been made by NEPI-Bot to create or otherwise instantiate this directory. In the absence of the official Data Directory, NEPI-Bot has no ability to secure the SDK’s Status and Data records.
3. assumes the existence of the Database Directory: *<NEPI\_home>/db*. Provision has been made, however, for the NEPI-Bot Subsystems to instantiate this directory if necessary but, to do so, requires that all running NEPI-Bot Subsystems have the necessary permissions.

*more assumptions coming …*

## Subsystem Configuration

When awakened by the Numurus SDK at appropriate, configurable times, Bot-Send configures itself by accessing the necessary information first from the singleton record in the ‘admin’ Table of the Float database, next from the “NEPI-Bot Configuration File,” or, as a last resort, utilizing internal, built-in default values. For details regarding the format, contents, defaults, and uses of the configuration, control, parameter, and attribute settings, see the “NEPI-Bot Subsystem Configuration” section above.

## Status and Data Retrieval

Subsequent to Subsystem instantiation and configuration at wake-up, Bot-Send’s first task is to determine the extent of the Status and Data Product collection (as performed by the Numurus SDK Application) since Bot-Send was last awakened.

As specified in the “NumSDK - NEPI-Bot ICD” (the source document for interface-related specifications between the Numurus SDK and NEPI-Bot Applications): “The interface is file-based, consisting of a set of predefined filesystem paths, predefined file formats, and basic application controls (presented as executable shell and/or Python scripts) available to the NumSDK and NEPI-Bot for communication and coordination. Additional coordination and communication capabilities are generally restricted to [the] existence or non-existence of various filesystem nodes.”

All SDK-Bot API-related files are JSON-formatted and, where appropriate, these file formats will be reprised in this document or expanded upon as necessary to support NEPI-Bot Application requirements.

Per details in the “NumSDK - NEPI-Bot ICD,” all collected Status and Data Product information is stored in the Float’s official “Data Directory” (*<NEPI-Home>/data*) which may be referred to as the “Data Upload Folder” in other Numurus Float Device documents. In this Data Directory are sub-directories (*i.e.*, one “Data Folder” for each Numurus SDK sampling) with the name formatted as *YYYY\_MM\_DD\_HHMMSS.sss*. (see Figure 3 below for a sample Data Directory with Data Folders).

**Bot-Send**

**2019\_02\_01\_120734.789**

**. . .**

**2019\_02\_01\_120202.456**

**2019\_02\_01\_120101.123**

**2019\_02\_01\_120000.000**

**all\_data\_dirs =**

**<*NEPI\_home*>/*data*/**

**2019\_02\_01\_120000.000/**

**2019\_02\_01\_120101.123/**

**2019\_02\_01\_120202.456/**

**. . .**

**2019\_02\_01\_120734.789/**

**SD Card**

Data Folder List

Figure 7: Bot-Send List of “Data Folders” in the “Data Directory” (created by NumSDK)

The Data Folder names are retrieved from SD card storage and retained in a Python list. Although not germane to subsequent Status and Data Product evaluation or outbound transmission management, the list in sorted in descending order for internal logging and debugging purposes.

**<*NEPI\_home*>/*data*/*2019\_02\_01\_120202.456/***

**Bot-Send**

**sys\_status.json**

**ACL\_1\_meta.json**

**ACL\_1\_std.json**

**ACL\_2\_meta.json**

**ACL\_2\_std.json**

**…**

**NDX\_1\_meta.json**

**NDX\_1\_chg.json**

**NDX\_1\_change.json**

**“Data Directory”**

**“Data Folder”**

**Status File and**

**Data Product File Sets**

Figure 8: Sample Status and Data Product Files in a Typical "Data Folder"

Each Data Folder contains *exactly* one mandatory, JSON-formatted “Status” file (using a configurable filename as specified in the Bot Configuration File; see above) that represents the overall system status information common to all Data Product file sets in that Data Folder (*i.e.*, at the time of the sampling).

In addition to the mandatory Status File, a Data Folder may contain *n*-number of “Data Products” (*n* may be 0) where each Data Product is comprised of a “set” of 2 or 3 files. Each Data Product file set has a single, mandatory, JSON-formatted “metadata file,” named *<Node\_ID>\_meta.json*, which contains the essential information about its Data Product file. In addition to the metadata file, a mandatory “Standard” data file (*<Node\_ID>\_std.json*) and an optional “Change” data file (*<Node\_ID>\_change.json*) comprise the balance of Data Product file set. See the “NumSDK - NEPI-Bot ICD” for a detailed description of the Status File and various Data Product file sets, extensions, and their contents.

Bot-Send processes each Data Folder by first consuming the single Status file (*sys\_status.json*) and inserting the Status file’s information - along with additional internal control fields - into the ‘status’ Table (see Table 3 below) of the Float’s embedded database. As the Status file is stored in the database, Bot-Send retains the unique “Row ID” (the “primary key”) of the stored record (see Figure 5 below).

**“Data Folder”**

**…/*2019\_02\_01\_120202.456/***

**sys\_status.json**

**ACL\_1\_meta.json**

**ACL\_1\_std.json**

**ACL\_2\_meta.json**

**ACL\_2\_std.json**

**…**

**NDX\_1\_meta.json**

**NDX\_1\_chg.json**

**NDX\_1\_change.json**

***float.db***

**Bot-Send**

Figure 9: Bot-Send Processing of the Data Product File Set

Subsequent to processing the unique Status file, each “metadata” file (*<\*>\_meta.json*) is processed, during which time, the mandatory “Standard” file (*<\*>\_std.json*) and optional “Change” file (*<\*>\_change.json*) is evaluated (*e.g.*, regarding their “sizes”). As with the Status file, the “metadata” file information is stored in the ‘data’ Table (see Table 4 below) of the Float’s embedded database. As the “metadata” file’s information is stored in the database, the associated Status record’s unique “Row ID” (saved from before) is added to the Data record as a “pointer” back to its “parent” Status record.

## Persistent Storage (Database Manager: *botdb.py*)

### Introduction

Effective and efficient management of the Float’s information (e.g., Status and Data Product records, PIPO analysis, archiving, housekeeping, etc.), necessitates the implementation of a small-footprint, embedded relational database. SQLite has been selected for inclusion in the Float’s development effort.

SQLite is available as an in-process Python library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private.

The *botdb.py* class library has been designed to provide a simple, DB-agnostic database instantiation, table and index creation, record insertion and update management, record deletion and purging, and common housekeeping duties. The SQLite database engine is the current implementation and has the ability to create in-memory databases for use in special cases as required.

[ Note: At the present time, the Float’s Database is used only by the NEPI-Bot Application. In the future, other Applications (*e.g.*, the Numurus SDK) may elect to interact directly with the Float’s Database and the implementation has been designed with that in mind. In all cases, the botdb.py class library will provide the sole API for access to the embedded SQLite database. ]

### Database Instantiation

The Float’s embedded database (*float.db*) is located in the *<NEPI\_home>/db* Database Directory. While NEPI-Bot assumes the existence of the official “Database Directory,” provision has been made for NEPI-Bot to instantiate the directory if necessary. This may be required when the database itself is created during “first start-up” or in the eventuality that a database reset is requested from “the Cloud.” In order for any of the NEPI-Bot Subsystems to make the official Float Directory in the NEPI Home directory, they must be granted the necessary permissions.

There is no requirement for the Float’s database file to exist prior to deployment. The primary NEPI-Bot Applications, Bot-Recv and Bot-Send, have the ability to check on start-up for the existence of the Float’s Database: a file called *<NEPI\_home>/db/float.db*. If the Float’s Database file does not exist, the first Application to notice its absence will create the embedded Database immediately and instantiate all necessary Tables (i.e., the ‘admin,’ ‘status,’ and ‘data’ Tables) and all associated Indexes. This same create capability exists to support a “database reset” Configuration and Control Message from “the Cloud.”

### Admin Table (‘*admin’*)

Although still supported, active use of the ‘admin’ Table has been suspended until future releases where additional features and capabilities are anticipated (especially in commercial applications).

The ‘admin’ Table contains the Bot’s entire suite of Configuration settings, along with a number of tracking and control elements necessary for efficient and effective Application operations.

Of interest is the fact that the ‘admin’ Table contains only a single record, created when the embedded database is instantiated (either at first Bot-Recv/Bot-Send Application start-up or due to a “Database Reset” Configuration and Control message from “the Cloud.” If the Float’s database exists at Application start-up, this unique record is selected from the ‘admin’ Table and used as the exclusive source for the Application’s configuration and control directives.

The ‘admin’ Table record is seeded with the contents of the Bot Configuration File and default values for various internally-used fields for Application control. For data consumed directly from the Configuration File, only the field name had been included in the Description/Notes column below. In the case of additional control fields, more detailed information has been included.

For convenience, a number of ‘columns’ have been added to the ‘admin’ Table that provide

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **‘admin’ Table**  **Column Name** | **JSON Type** | **Python**  **Type** | **SQLite Type**  **Affinity** | **Range/Format** | **Description/Notes** |
| *row\_id*  **(hidden)** | - | integer | BIGINT  (INTEGER) | **[ 1 - 263 ]** | **Admin Record Index**  (*implicit* autoincrement column; 64-bit signed integer; only 1 record in this Table with rowid=1) |
| *machine* | number | integer | TINYINT  (INTEGER) | **[ 0 - 3 ]** | **Machine Type** |
| *debugging* | number | bool | BOOLEAN  (NUMERIC) | **[ True, False ]** | **Debug Mode** |
| *logging* | number | bool | BOOLEAN  (NUMERIC) | **[ True, False ]** | **Logging Mode** |
| *locking* | number | bool | BOOLEAN  (NUMERIC) | **[ True, False ]** | **Locking Mode** |
| *host* | string | string | TEXT  (TEXT) | **Class A, B, or C IP Addresses**  Factory Default:  ***10.0.0.116 (ALPHA)***  ***18.223.151.16 (TEST)***  ***0.0.0.0 (FLOAT)*** | **Host IP Address** |
| *port* | number | integer | SMALLINT  (INTEGER) | **[ n ]**  Factory Default:  ***1024 (ALPHA)***  ***1024 (TEST)***  ***370 (FLOAT)*** | **Port** |
| *protocol* | number | integer | TINYINT  (INTEGER) | **[ 0 - 4 ]** | **Comm Protocol** |
| *packet\_size* | number | integer | SMALLINT  (INTEGER) | **[ n ]**  Factory Default:  ***1024 (ALPHA)***  ***1024 (TEST)***  ***370 (FLOAT)*** | **Protocol Packet Size** |
| *sys\_status\_file* | string | string | TEXT  (TEXT) | **<*file\_name*>.json**  Factory Default: ***sys\_status.json*** | **Status File Name** |
| *data\_dir* | string | string | TEXT  (TEXT) | **<*relative\_path*>**  Factory Default: ***data*** | **Data Directory** |
| *data\_dir\_path* | string | string | TEXT  (TEXT) | **<*relative\_path*>**  Factory Default: ***log*** | **Data Directory Path**  (full path to the Float’s Data Directory; saves recalculating in later Application executions) |
| *log\_dir* | string | string | TEXT  (TEXT) | **[ 0.0 -100.0 ]** | **Log Directory** |
| *br\_log\_name* | string | string | TEXT  (TEXT) | **<*file\_name*>**  Factory Default: ***brlog.txt*** | **Bot-Recv Log File Name** |
| *br\_log\_file* | string | string | TEXT  (TEXT) | **<*full\_path\_to\_file*>** | **Bot-Recv Log File**  (full path of the Bot-Recv Log File; saves recalculation in later App executions) |
| *bs\_log\_name* | string | string | TEXT  (TEXT) | **<*file\_name*>**  Factory Default: ***bslog.txt*** | **Bot-Send Log File Name** |
| *bs\_log\_file* | string | string | TEXT  (TEXT) | **<*full\_path\_to\_file*>** | **Bot-Send Log File**  (full path of the Bot-Recv Log File; saves recalculation in later App executions) |
| *wt\_changed* | number | bool | BOOLEAN  (NUMERIC) | **[ True | False ]**  Factory Default: ***False*** | **Weight Factor Changed** |
| *pipo\_scor\_wt* | float | float | FLOAT  (REAL) | FLOAT  (REAL) | **Score Weight Factor** |
| *pipo\_qual\_wt* | float | float | FLOAT  (REAL) | **[0.0 - 1.0]**  Default: **0.5** | **Quality Weight Factor** |
| *pipo\_size\_wt* | float | float | FLOAT  (REAL) | **[0.0 - 1.0]**  Default: **0.5** | **Size Weight Factor** |
| *pipo\_trig\_wt* | float | float | FLOAT  (REAL) | **[0.0 - 1.0]**  Default: **0.5** | **Trigger Weight Factor** |
| *pipo\_time\_wt* | float | float | FLOAT  (REAL) | **[0.0 - 1.0]**  Default: **0.5** | **Time Weight Factor** |
| *purge\_rating* | float | float | FLOAT  (REAL) | **[0.0500 - 0.9000]**  Default: ***0.1000*** | **Purge Rating** |
| *max\_msg\_size* | integer | integer | SMALLINT  (INTEGER) | **[1024-n]**  Default: ***65536*** | **Max Message Size** |
| *reserved1* | - | - | BLOB  (BLOB) | - | **-** |
| *reserved2* | - | - | BLOB  (BLOB) | - | **-** |
| *reserved3* | - | - | BLOB  (BLOB) | - | **-** |
| *reserved4* | - | - | BLOB  (BLOB) | - | **-** |

Table 4: The 'admin' Table Schema (with Configuration File Mappings)

### Status Table (‘*status’*)

A single, JSON-formatted Status file (*sys\_status.json*) is consumed from each Data Folder discovered by the Bot-Send subsystem (*i.e.*, one “Data Folder” for each Numurus SDK sampling). The file is parsed, and the Value associated with each keyword is stored in its respective Status Table “Column.”

The Status Table in the Float’s database is a so-called “rowid” Table, which is distinguished by the fact that it has a unique, non-NULL, signed 64-bit integer that is used as the access key for the “records” in the underlying B-tree storage engine. Access to records via rowid is highly optimized and very fast. The rowid is created automatically as each Status record is inserted into the Table. This rowid also acts as a “foreign key” in the Data Table records (see the “Data Table” section later in this document), “linking” individual Data Table records back to their associated Status record here in the Status Table.

When a Status record is created, the “state” Column id seeded with a ‘0’ value, indicating that the record is in the “created” state. The “state” Column is updated periodically during the Status record’s lifetime to indicate, further, that the Status record’s information has been sent to “the Cloud,” has been acknowledged as received, has been targeted for purging during housekeeping, and/or other necessary contextual applications.

At Status record creation, the “base” Column is seeded with a ‘0’ value, indicating that this record forms part of the “base set” of records, to which “the Cloud” can be “reset” if such a Request is made from “the Cloud.” A base record is retained in the database indefinitely but can be replaced at any time should newer base records supersede it (at which point it will likely be targeted for deletion/purging).

At the time of this writing, the Status Table contains four (4) special Columns (reserved[1-4]) reserved for future expansion and/or enhancement. They are created as NULL BLOBs in this revision but are easily transformed as required.

The remaining Status Table Columns are seeded at record creation with the Values from the keyword-value pairs in the JSON-formatted Status file according to the types, ranges, formats, and descriptions presented in Table 2 below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Column Name** | **JSON Type** | **Python**  **Type** | **SQLite Type**  **Affinity** | **Range/Format** | **Description/Notes** |
| *row\_id*  **(hidden)** | - | integer | BIGINT  (INTEGER) | [ 1 - 263 ] | **Status Record Index**  (*implicit* autoincrement column; 64-bit signed integer) |
| *state* | - | integer | TINYINT  (INTEGER) | [ 0 - 3 ] | **Status Record State**  (0=active; 1=sent; 2=ack) |
| *base* | - | bool | BOOLEAN  (NUMERIC) | True, False | **Base Record**  (forms part of the “last” deliverable base of Status and Data Product records) |
| *meta\_state* | - | integer | TINYINT  (INTEGER) | [0-TBD] | **Meta State**  (unused through release 6/1) |
| *reserved1* | - | - | BLOB  (BLOB) | - | **-** |
| *reserved2* | - | - | BLOB  (BLOB) | - | **-** |
| *timestamp* | number | float | DOUBLE PRECISION  (REAL) | [ (0.0 - 232) ] | **Unix Epoch Time**  (millisecond resolution) |
| *serial\_num* | string | string | TEXT  (TEXT) | TBD | **Serial Number**  (specifies the Float’s Serial Number) |
| *sw\_rev* | string | string | TEXT)  (TEXT) | TBD | **Software Revision**  (specifies overall s/w revision for the system; independent of config index values) |
| *sw\_rev\_increment* | - | integer | TINYINT  (INTEGER) | TBD | **Software Revision Increment**  (specifies incremental release number) |
| *navsat\_fix\_time* | number | float | DOUBLE PRECISION  (REAL) | [ 0 - 232 ] | **Unix Epoch Time**  (last satellite fix) |
| *latitude* | number | float | DOUBLE PRECISION  (REAL) | [ -90.0 - 90.0 ] | **Latitude Position**  (positive is north of equator) |
| *longitude* | number | float | DOUBLE PRECISION  (REAL) | [ -180.0 - 180.0 ] | **Longitude Position**  (positive is east of prime meridian) |
| *heading* | number | float | DOUBLE PRECISION  (REAL) | [ 0.0 - 360.0 ] | **Float Heading**  (magnetic north) |
| *batt\_charge* | number | float | FLOAT  (REAL) | [ 0.0 -100.0 ) | **Battery Charge**  (percentage charge remaining) |
| *bus\_voltage* | number | float | FLOAT  (REAL) | TBD | **Main Bus Voltage** |
| *temperature* | number | float | FLOAT  (REAL) | TBD | **Main Temperature** |
| *trig\_wake\_count* | number | integer | SMALLINT  (INTEGER) | [ 0 - 10000 ] | **Trigger Wake Count**  (number of times Zynq has awoken due to sensor detection triggers) |
| *wake\_event\_type* | number | integer | TINYINT  (INTEGER) | 0: alarm, 1: trigger | **Wake Event Type**  (specifies the event that initiated the data collection) |
| *wake\_event\_id* | number | integer | SMALLINT  (INTEGER) | *task\_id* for alarms, *smarttrigger\_id* for trigger | **Wake Event ID**  (interpretation depends on value of the *wake\_event\_type*) |
| *task\_index* | number | integer | SMALLINT  (INTEGER) | [ 0 - 255 ] | **Task Index**  (most recent task schedule update index) |
| *trig\_cfg\_index* | number | integer | SMALLINT  (INTEGER) | [ 0 - 255 ] | **Trigger Config Index**  (most recent SmartTrigger configuration index) |
| *rule\_cfg\_index* | number | integer | SMALLINT  (INTEGER) | [ 0 - TBD ] | **Rule Config Index**  (most recent SmartTrigger rule modification index) |
| *sensor\_cfg\_index* | number | integer | SMALLINT  (INTEGER) | [ 0 - TBD ] | **Sensor Config Index**  (most recent sensor configuration index) |
| *node\_cfg\_index* | number | integer | SMALLINT  (INTEGER) | [ 0 - TBD ] | **Node Config Index**  (most recent Node configuration index) |
| *geofence\_cfg\_index* | number | integer | SMALLINT  (INTEGER) | [ 0 - TBD ] | **Node Config Index**  (most recent Node configuration index) |
| *state\_flags* | number | integer | SMALLINT  (INTEGER) | 32-bit mask. Contents TBD. | **State Flag Mask**  (*e.g.*, temperature and storage warnings, drag line state, etc.) |
| *reserved3* | - | - | BLOB  (BLOB) | - | **-** |
| *reserved4* | - | - | BLOB  (BLOB) | - | **-** |

Table 5: The ‘status’ Table Schema (with Status File Mappings)

### Data Product Table (‘*meta’*)

The SDK’s JSON-formatted Data Product samplings are split into three possible files: 1) the primary “data” file *(<Node\_ID>\_meta.json*), 2) a mandatory “standard” file (*<Node\_ID>\_std.json*) and, 3) an optional “change” file (*<Node\_ID>\_change.json*).

Each Data Product’s metadata file is parsed, and the Values associated with each keyword are stored in their respective Float database “columns.”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Column Name** | **JSON Type** | **Python**  **Type** | **SQLite Type**  **(Affinity)** | **Range/Format** | **Description/Notes** |
| *row\_id*  **(hidden)** | - | integer | BIGINT  (INTEGER) | **[ 1 - 263 ]** | **Data Record Index**  (*implicit* autoincrement column; 64-bit signed integer) |
| *state* | - | integer | TINYINT  (INTEGER) | **[ 0 - 7 ]** | **Record State**  (0=created; 1=packed; 2=sent; 3=ack; 4=deleted, etc. *TBD* |
| *base* | - | bool | BOOLEAN  (NUMERIC) | **[ True | False ]** | **Base Record**  (forms part of the “last” deliverable base of Status and Data Product records) |
| *status* | - | integer | BIGINT  (INTEGER) | **[ 1 - 263 ]** | **Status Record ID**  **[foreign key]**  (a pointer to the ‘*row\_id*’ associated System Status Record in the Status Table) |
| *numerator* | - | float | FLOAT  (REAL) | **[ 0.0 - 2.0 ]** | **PIPO Numerator**  (the current, calculated numerator of this record’s PIPO formula) |
| *trigger* | - | float | FLOAT  (REAL) | **[ 0.5 | 1.0 ]** | **Trigger Value**  (from a lookup table; a combination of the *wake\_event\_type* and *wake\_event\_id* values from the associated sys\_status.json file; used in prioritizing data) |
| *pipo* | float | float | FLOAT  (REAL) | **[ 0.0 - 2.0 ]** | **PIPO Score**  (the current, calculated PIPO score for this Data Product) |
| *type* | string | string | CHARACTER(3)  (TEXT) | **[ [A-Z][A-Z][A-Z] ]** | **Data Type ID**  <t>\_**<i>**\_data.json  <t>\_**<i>**\_std.json  <t>\_**<i>**\_change.json  (a 3-char string <t>, parsed from the Data Product’s file name, identifying the ”type” of Data Product) |
| *instance* | integer | integer | TINYINT  (INTEGER) | **[ 0 - 15 ]** | **Data Instance**  <t>\_**<i>**\_data.json  <t>\_**<i>**\_std.json  <t>\_**<i>**\_change.json  (a number <i>, parsed from the Data Product’s file name, identifying the ”instance” of this Data Product’s production) |
| *timestamp* | float | float | FLOAT  (REAL) | **Unix Epoch Timestamp**  (Millisecond resolution) | **Create Time**  (timestamp of the start of data collection. May differ from containing folder’s timestamp due to differences in when *sys\_status.json* info is captured and when data is acquired) |
| *heading* | number | float | DOUBLE PRECISION  (REAL) | **[ 0.0 - 360.0 ]** | **Float Heading**  (the current reading; magnetic north) |
| *quality* | number | integer | TINYINT  (INTEGER) | **[ 0 - 100 ]** | **Quality Value**  (a node-dependent indicator of quality; used in prioritizing data) |
| *score* | number | float | FLOAT  (REAL) | **[ 0.0 - 1.0 ]** | **Score Value**  (a node-dependent score value; used in prioritizing data) |
| *metafile* | string | string | TEXT  (TEXT) | ***<Node\_ID>\_meta.json*** | **Meta Data File**  (same <Node\_ID> as this file; NULL if no binary data for this node type) |
| *stdfile* | string | string | TEXT  (TEXT) | ***<Node\_ID>\_std.json*** | **Standard Data File**  (same <Node\_ID> as this file; NULL if no binary data for this node type) |
| *chgfile* | string | string | TEXT  (TEXT) | ***<Node\_ID>\_change.json*** | **Change Data File**  (same <Node\_ID> as this file; NULL if no binary data for this node type) |
| *stdsize* | - | integer | INTEGER  (INTEGER) | **[ 1 - *n* ]** | **Standard Data File Size**  (file size in bytes; used for PIPO calculations; saves re-reading SD card on recalculations forced by config changes.) |
| *chgsize* | - | integer | INTEGER  (INTEGER) | **[ 1 - *n* ]** | **Change Data File Size**  (file size in bytes; used for PIPO calculations; saves re-reading SD card on recalculations forced by config changes.) |
| *norm* | - | float | DOUBLE PRECISION  (REAL) | - | **Size Normalization** |
| *std* | - | string | TEXT  (TEXT) | - | **Standard File** |
| *chg* |  | string | TEXT  (TEXT) |  | **Change File** |
| *chg\_eligible* |  | bool | BOOLEAN  (NUMERIC) |  | **Change File Eligible** |
| *pitch* |  | integer | INTEGER  (INTEGER) |  | **Float Pitch** |
| *roll* |  | integer | INTEGER  (INTEGER) |  | **Float Roll** |
| *channels* |  | integer | INTEGER  (INTEGER) |  | **Sample Channels** |
| *rows* |  | integer | INTEGER  (INTEGER) |  | **Sample Rows** |
| *columns* |  | integer | INTEGER  (INTEGER) |  | **Sample Columns** |
| *samples* |  | integer | INTEGER  (INTEGER) |  | **Sample Count** |
| *dtype* |  | string | TEXT  (TEXT) |  | **Data Type** |
| *data* |  | string | TEXT  (TEXT) |  | **Data** |
| *chg\_channels* |  | integer | INTEGER  (INTEGER) |  | **Change Channels** |
| *chg\_dtype* |  | string | TEXT  (TEXT) |  | **Change Data Type** |
| *chg\_deltas* |  | string | TEXT  (TEXT) |  | **Change Deltas (Data)** |
| *node\_code* |  | integer | INTEGER  (INTEGER) |  | **Node Code** |

Table 6: The ‘data’ Table Schema (with Data File Mappings)

## Data Product Prioritization (PIPO Manager: *botpipo.py*)

### Introduction

The Priority-In Priority-Out (PIPO) Manager’s job is to consume all folders containing SDK-created Data Products, store the discovered status and data records in the Float’s embedded database, prioritize the data records (*i.e.*, calculate their respective PIPO Scores), and, with aging as a factor, purge all status and data records that eventually fall below a minimum PIPO Score.

### Status and Data Product Consumption

For each Data Folder in the Float’s Data Directory (used by the Numurus SDK to provide its Status and Data information to the Bot-Send subsystem), all JSON-formatted files contained therein (*i.e.*, a Status file and *n*-number of Data Product files associated with that Status file) are consumed by the PIPO Manager. The Status and Data file formats and contents are described in the “NumSDK - NEIP-Bot ICD” document.

All consumed Status and Data records are inserted into corresponding Tables in the Float’s embedded database, based on the schema described in the “Persistent Storage” section of this document (*i.e.*, the Database Manager: *botdb.py*). Additional fields (*i.e.*, columns) have been added to the Status and Data Tables in the Float’s embedded database to accommodate various caching and control functionality implemented by Bot-PIPO’s operational features, such as PIPO Ratings, subsequent Status and Data record selection, Status and Data record association (*e.g.*, foreign keys), database housekeeping, etc.

Upon successful insertion of all Status and Data Product file information into the Float’s embedded database, the PIPO Manager purges (deletes) the associated Data Folder in the Data Directory.

### Data Product Prioritization

Bot-PIPO, a NEPI-Bot library component, acts as the Priority-In Priority-Out (PIPO) Manager. Its job is to take all newly-created Data Products (*i.e.*, those Data Products captured and stored by the Numurus SDK since NEPI-Bot’s last “wake-up”) and compute a “Priority Score” for each one. In addition, Bot-PIPO re-computes the Priority Scores for all archived records *(i.e.*, those Data Products previously captured by the Numurus SDK during earlier wake cycles but, due to lower PIPO ratings, were not uploaded to the surface and, instead, saved in the Float’s DB to be re-evaluated for possible future upload).

Data Product “Priority Ratings” are calculated based on a prescribed, configurable formula, employing the following characteristics:

PIPO Rating = (A\*Score + B\*Quality + C\*Size + D\*Trigger) / ((E\*Age) + 1)

where:

* A, B, C , and D, are “Weighting Factors,” ranging from [ 0.0 <= (A,B,C,D) <= 1.0 ],
* Additionally, 1-ε <= A2 + B2 + C2 + D2 < 1+ε (ε represents a small ‘epsilon’ variation),
* Values for ‘Score,’ ‘Quality,’ ‘Size,’ and ‘Trigger,’ are each normalized to [ 0.0 <= V <= 1.0 ],
* E is a Weighting Factor, ranging from [ 0.0 <= E < ∞ ], and
* Value for ‘Age’ ranges from [ 0.0 <= V < ∞ ]

All Weight Factors are resident in the Bot Configuration File (A=*pipo\_scor\_wt*, B=*pipo\_qual\_wt*, C=*pipo\_size\_wt*, D=*pipo\_trig\_wt*, and E=*pipo\_time\_wt*) and are Cloud-configurable using the associated Control and Configuration Messages.

The E weighting factor is unrestricted and gives control over the age-based decay rate. The Data Product’s “Age” represents a fixed time unit represented in hours. Technically, it doesn't matter which time unit is used (*i.e*., since the E Weighting Factor will scale it rationally), but “hours” unit has been selected as being a reasonable representation within the scope of a limited Float lifespan. The addition of “1” to the E\*Age multiplication in the formula’s denominator avoids any possibility of division by zero (0). These conditions assure that the PIPO Rating retreats asymptotically toward 0 for any Data Product and is, therefore, guaranteed to drop below the Purge Threshold (see below) at some point.

Once calculated by the PIPO Manager, the “numerator” value (in the PIPO Rating formula) is stored by the PIPO Manager in the Float’s embedded database along with the rest of the related Data Product’s information. This technique eliminates the need for recalculating a significant portion of the formula for unsent Data Products (i.e., those not “making the cut” for upload due to a low PIPO Rating as calculated in one or more previous wake cycles). “Numerator” values for previously-evaluated Data Records require recalculation only in the event of a configuration change to the A, B, C, and/or D Weighting Factors.

### Upload Message Buffer Creation

A “Upload Message Buffer” of configurable size (see: “*max\_msg\_size*” in the Bot Configuration File above) is created by the PIPO Manager to accommodate as much Status and Data information as possible, selected from the Float’s embedded database, based on a high-to-low priority score.

This buffer forms a single payload, subsequently sent to “the Cloud” by the *Bot-Comm* subsystem later in the *Bot-Send* wake-cycle. The buffer represents a data stream of information groups, where each grouping is comprised of a common Status record and *n*-number of Data Product records associated with that Status record.

### Data Product Selection

Data Product records are selected from the database, sorted in descending order, based on the records’ PIPO Ratings. The “results” from this database search are used to determine which Data Product records will be sent to “the Cloud.”

### Message Packetization

The Upload Message Buffer (described above) is loaded first with all Float Request Messages (see the “Request Messages” section in “Messaging Packaging (Message Manager: bot-proc.py)” below), followed by all Status and Data Product records that fit within the limits of the Upload Message Buffer.

This section describes the *process* of packetization only. It must be noted that all Status and Data Product records that are “packed” into the Upload Message Buffer via this process are subjected to the “bit-packing” and “compression” processes described in the “Messaging Packaging (Message Manager: *bot-proc.py*)” section below.

In the Float’s embedded database, all Data Product records consumed from the same Data Folder in the Data Directory are associated with the sole Status record, resident in that same Data Folder (using “foreign keys). This relationship is important in that Data Product records from any given Data Folder may not demonstrate similar PIPO Ratings and some, in fact, may not have a PIPO Rating sufficient to “make the cut” for transmission to “the Cloud” in the same Upload Message.

During the packetization process, when a Data Product record is chosen for inclusion in the Upload Message Buffer, a check is made to see if the associated Status record has: 1) been previously-included in the current message buffer or 2) been previously-uploaded to “the Cloud” in an earlier cycle’s Upload Message Buffer. If neither condition is true, the Status record is packed into the current Upload Message Buffer, followed immediately by the associated Data Product record. If the associated Status record has been included in the current buffer or previously uploaded to “the Cloud,” only the Data Product record is packetized into the current Upload Message Buffer.

The Status and Data record packetizing process continues until such time as the inclusion of a Data Product record (with or without its associated Status record) would overflow the limits of the Upload Message Buffer, at which point the offending Data Product record is simply ignored.

### Data Product Purging

Data Products not included in a wake-up cycle’s upload to the Cloud, primarily due to PIPO scores that fall below “the cut-off rating” for that cycle, are retained in the Float’s embedded database for future upload consideration. To protect the Float’s limited storage capacity, a “Purge Threshold” (i.e., a configurable, lower bound for a Data Product’s PIPO Rating) has been implemented.

The “Purge Threshold” (‘*purge\_rating’*) is found in the Bot Configuration File and represents the minimum PIPO Rating, below which a Data Product will be purged (deleted) from the Float’s embedded database. As indicated above, the PIPO formula’s vectoring constraints assure that all Data Product ratings retreat asymptotically toward 0 and is, therefore, guaranteed to drop below the Purge Threshold at some point.

The following purge model applies:

1. If a Data Product has PIPO\_Rating < *purge\_rating*, it is deleted from the DB.
2. If a Data Product has been uploaded successfully to the Cloud, it is deleted from the DB.
3. If a Status message has been uploaded successfully to the Cloud, it is deleted from the DB.

## Uplink Message Buffer Packaging (Message Manager: *botmsg.py*)

### Introduction

The Bot-Msg Class Library is designed to provide total control over uplink message construction and formatting. The Class Library allows for the instantiation of a Bot Message Object along with two important Class Methods: packstatus() to pack a Status Record into an Uplink Message and packmeta() to pack a Data Product into an Uplink Message (which includes the Data Product’s Meta Data plus either the ‘Standard’ or “Change’ data, based on the Change Data’s eligibility).

An Uplink Message is comprised of

### Message Creation

The Bot-Msg Class Object is instantiated at an appropriate time in Bot-Send’s execution cycle, usually near the beginning of the process, using the following coding example:

from botmsg import BotMsg

...

sm = BotMsg(cfg\_object *cfg*, log\_object *log*, int *log\_level*)

where the ‘*cfg*’ and ‘*log*’ objects are those instantiated earlier in a subsystem’s execution and the ‘*log\_level*’ represents the debugging and logging depth (and indentation) desired.

Unlike some of the other NEPI-Bot Class Libraries, BotMsg needs no call to a subsequent ‘initialization’ Method to ready its object for use.

### Status Messages

A Status Record is derived from information originating in its respective SDK-created *sys\_status.json*. Status Files accumulate in the Float’s Data Folders as described in the “Numurus SDK - NEPI-Bot ICD.” Each JSON-formatted Status File is consumed by the Bot-Send Application and the associated data information - along with additional control elements - is inserted as a single record into the ‘status’ Table in the *float.db* SQLite3 database.

Each Status Message reflects the following general format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Msg Header** | **Status Record 1** | **Status Record 2** | **Status Record 3** | **…** | **Status Record *n*** |

Figure 10: General Layout of a Bot-Mess Uplink Status Message.

The Status Message “Msg Header” contains the following information:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Msg Length** | **Reference Time** | **Serial Number** | **Software Revision** |
| *bits 0-7* | *bit 8-31* | *bits 32-63* | *bits 64-159* | *bits 160-239* |

Figure 11: Layout of the “Msg Header” om the Bot-Mess Uplink Status Message

The Status Message Header is comprised of thirty (30) bytes of information (described in the Table below) and applies in varying ways to all Status Records contained in the uplink Status Message. All fields in the Msg Header are guaranteed to be formatted on byte boundaries with the least significant bit of each field positioned to the right; zeros are left-padded as necessary.

While the “Type” field is a full character, the left-most five (5) bits are reserved for future use and, at this writing, are guaranteed to be zero-filled.

The “Msg Length” field is three (3) bytes and provides for a sufficiently large integer value to accommodate a Status Message length of up to 16MB (although a message of that size is unlikely. In future hardware and software versions, some of the more-significant bits may be pirated for other uses.

The “Reference Time” is a standard 32-bit UNIX Epoch Time and provides a referential base that is added to timestamps in the following Status Records to yield the *actual* time intended by those fields. The Reference Time is guaranteed to represent a time less than any time presented in the Status Records. By employing this method, timestamp fields in the Status Record can be stored in twenty-four (24) bits and, after unpacking, adding the Reference Time to the unpacked field will provide that real sampled time.

The “Serial Number” and “Software Revision” are mostly-static values but should be used to ensure that Status Records are recorded with this information in the eventuality that these two identifiers change during the Float’s expected lifetime (an unlikely event, but possible).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Component Name**  **(included in)** | **Bot**  **Type** | **Range/Format** | **Req**  **Bits** | **Msg**  **Bits** | **Notes** |
| *type* | integer | bits 0-4: reserved  bits 5-7: [0-3] | 8 | 0-7 | **Message Type**  (least significant 3 bits used to identify message type; stored on byte boundary; 1=Request, 2=Status, 3=Data) |
| *length* | integer | [ 0 - 224-1 ] | 24 | 8-31 | **Message Length**  (total bytes in Status Message, including the header and before compression) |
| *reference* | integer | Unix Epoch Time | 32 | 32-63 | **Reference Time**  (guaranteed to represent a point in time “just prior” to any Status Record timestamps included in the current Status Message) |
| *serial\_num*  (header) | string | AA\_AAA\_NNNNN  (where A=[A-Z] and N=[0-9]) | 96 | 64-159 | **Serial Number**  (specifies the unique serial number for the Float; may be sent once in Status Message Header) |
| *sw\_rev*  (header) | string | NNNNNNNN\_N  where n=[0-9] | 80 | 160-239 | **Software Revision**  (specifies overall software revision for the system, independent of configuration index values; may be sent once in Status Message Header) |

Table 7: Bit Map of the Status Message "Msg Header"

The first character (char 0, bits 0-7) of the “Msg Header” contains 5 bits (0-4) held in reserve and a 3-bit “Msg Type” (5-7) designating the type of Message: 1=*Request*, 2=*Status*, 3=*Data*.

The Status Message “Msg Length” is contained in characters 1-3 (bits 8-31) and provides for lengths up to 16MB (somewhat beyond what is anticipated during deployment).

The Status Message “Reference Time” is contained in characters 4-7 (bits 32-63) and represents a 32-bit UNIX timestamp (UNIX Epoch time; the number of seconds that have elapsed since 00:00:00 Thursday, 1 January 1970, Coordinated Universal Time (UTC), minus leap seconds). The Reference Time is guaranteed to represent a point in time “just prior” to any Status Record timestamps included in the current Status Message. Timestamps within a Status Record are the difference between the actual recording of that timestamp and the Reference Timestamp.

### Status Message Bit-Packing

Following the Msg Header are n-number of 42-byte Status Records, each one bit-packed according to the Table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bot Uplink Msg Status Segment**  **Bit Map** | | **Element Name** | **Range/Format** | **Description/Notes** |
| **Bytes 0-3 (32-bit integer)** | **0-2**  (3 bits) | *segid* | [ 0 - 7 ] | **Segment ID**  (identifies the segment type; 0=reserved, 1=status, 2=meta, etc.; higher numbers reserved primarily for upload requests, alarms, etc.) |
| **3-15**  (13 bits) | *statid* | [ 0 - 8191 ]  ***mod 2\*\*13*** | **Status Record ID**  (the ‘primary key’ or ‘rowid’ that uniquely identifies this Status Record in the ‘status’ Table of the Float’s embedded database; this built-in SQLite field can range from 0 - 264-1 on the Float; must anticipate rollover) |
| **16-31**  (16 bits) | *heading* | [ 0 - 36,000 ]  ***centidegrees*** | **Heading**  (magnetic north) |
| **Bytes 4-11 (64-bit integer)** | **0-24**  (25 bits) | *timestamp* | Unix Epoch Time  ***1 sec resolution*** | **Timestamp**  (time when data collected; dated from Jan. 1 of the current year; must anticipate 1-year rollover) |
| **25**  (1 bit) | *longitude\_sign* | [ 0 | 1 ] | **Longitude Sign**  (longitude sign bit; a “0” bit value indicates a positive value, east of the prime meridian; a “1” bit value indicates a negative value, west of the prime meridian, requiring multiplication by -1) |
| **26-53**  (28 bits) | *longitude* | ( 0 - 180,000,000]  ***microdegrees*** | **Longitude**  (this absolute longitude value is exclusive of the longitude sign bit described above; multiplication by -1 is required if the longitude sign bit is 1) |
| **54-63**  (10 bits) | *trig\_wake\_count* | [ 0 - 1023 ]  ***mod 1024*** | **Trigger Wake Count**  (represents the number of times the float has been awakened by a trigger; must anticipate rollover) |
| **Bytes 12-15 (32-bit integer)** | **0-6**  (7 bits) | *batt\_charge* | [ 0 -100 ]  ***1% resolution*** | **Battery Charge**  (percentage battery charge remaining; integer precision reported) |
| **7-1**  (1 bit) | *bus\_voltage\_sign* | [ 0 | 1 ] | **Bus Voltage Sign**  (voltage sign bit; a “0” bit value indicates a positive value; a “1” bit value indicates a negative value, west of the prime meridian, requiring multiplication by -1) |
| **8-12**  (5 bits) | *bus\_voltage* | [ 0 - 31 ]  ***decivolts*** | **Bus Voltage**  (main bus voltage; integer represents decivolts; multiplication by -1 is required if the bus voltage sign bit is “1”) |
| **13-19**  (7 bits) | *temperature* | [ 0 - 100 ]  ***1% resolution*** | **Main Temperature**  (main float temperature; integer precision reported) |
| **20-25**  (6 bits) | *node\_cfg\_index* | [ 0- inf ]  ***mod 64*** | **Node Cfg Index**  (most recent Node configuration index; must anticipate rollover; likely <= 255) |
| **26-31**  (6 bits) | *geofence\_cfg\_index* | [ 0 - inf ]  ***mod 64*** | **GeoFence Cfg Index**  (Most recent Geofence configuration index; must anticipate rollover; likely <= 255) |
| **Bytes 16-19 (32-bit integer)** | **0-7**  (8 bits) | *wake\_event\_id* | [ 0 - 255 ] | **Wake Event ID**  (interpretation depends on the value of the wake\_event\_type field; task\_id for alarms,  smarttrigger\_id for trigger) |
| **8-13**  (6 bits) | *task\_cfg\_index* | [ 0 - inf ]  ***mod 64*** | **Task Index**  (most recent task schedule update index; must anticipate rollover; likely <= 255) |
| **14-19**  (6 bits) | *trig\_cfg\_index* | [ 0 - inf ]  ***mod 64*** | **Trigger Cfg Index**  (most recent *SmartTrigger* configuration index; must anticipate rollover; likely <= 255) |
| **20-25**  (6 bits) | *rule\_cfg\_index* | [ 0 - inf ]  ***mod 64*** | **Rule Cfg Index**  (most recent *SmartTrigger* rule modification index; must anticipate rollover; likely <= 255) |
| **26-31**  (6 bits) | *sensor\_cfg\_index* | [ 0 - inf ]  ***mod 64*** | **Sensor Cfg Index**  (most recent sensor configuration index; must anticipate rollover; likely <= 255) |
| **Bytes 20-23** | **0-31**  (32 bits) | *state\_flags* | [ 0 - 232-1 ]  ***32-bit mask*** | **State Flags**  (Contents TBD; e.g., temperature and storage warnings, drag line state; stored as-is in 32-bits) |
| **Bytes 24-27 (32-bit integer)** | **0-3**  (4 bits) | *sw\_rev* | [ 0 - inf ]  ***mod 16*** | **Software Revision Number**  (specifies overall s/w revision for the system; independent of configuration index values; allows for an unlikely 16+ revisions; must anticipate rollover, however) |
| **4**  (1 bit) | *latitude\_sign* | [ 0 | 1 ] | **Latitude Sign**  (latitude sign bit; a “0” bit indicates a positive latitude value, north of the equator; a “1” bit indicates a negative latitude value, south of the equator, requiring multiplication by -1) |
| **5-31**  (27 bits) | *latitude* | ( 0 - 90,000,000 ]  ***microdegrees*** | **Latitude**  (this value is exclusive of the latitude sign bit described above; multiplication by -1 is required if the latitude sign bit is 1) |
| **Bytes 28-30 (24-bit integer)** | **0-2**  (3 bits) | *N/A* | N/A | **Reserved**  (the only free bits in this otherwise fully-packed message segment) |
| **3-20**  (18 bits) | *navsat\_fix\_time* | Unix Epoch Time  ***1 sec resolution*** | **Nav Sat Fix Time**  (time of last satellite fix; dated from Jan. 1 of the current year; must anticipate 1-year rollover) |
| **21-23**  (3 bits) | *wake\_event\_type* | [ 0 - 7 ] | **Wake Event Type**  (Specifies the event - trigger or alarm - that initiated the data collection; 0=alarm, 1=trigger, 2-TBD, 3=TBD, etc.) |

Table 8: NumSDK Status Record to Cloud Message Bit-Packing Map

### Data Messages

*TBD*

### Data Message Bit-Packing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Bot Uplink Message**  **ALL Data Segments**  **Bit Map (Pt. 1)** | | | **Element Name** | **Range/Format** | **Description/Notes** |
| **Bytes 0-3 (32-bits) for BOTH std amd chg data** | **0-2**  (3 bits) | | *segid* | [ 0 - 7 ] | **Segment ID**  (identifies the segment type; 0=reserved, 1=status, 2=data, etc.; higher numbers reserved primarily for upload requests, alarms, etc.; ***guaranteed*** to be first field in segment) |
| **3-12**  (10 bits) | **3**  (1 bit) | *dp\_timestamp\_sgn* | [ 0 | 1 ] | **Data Product “Timestamp” Sign Bit**  (sign bit for the Data Product timestamp; 0=positive; 1=negative) |
| **4-12**  (9 bits) | *dp\_timestamp* | Unix Epoch Time  ***1 sec resolution*** | **Data Product “Timestamp”**  (time when Data Product collected; delta +/- 511 from Status Message ‘timestamp;’ must apply *dp\_timestamp\_sgn* above; cloud must anticipate 1-year rollover) |
| **13-18**  (6 bits) | | *dp\_node\_code* | [ 0 - 63 ] | **Data Product “Node Code”**  (equivalent to the SDK’s “Node Type ID;” integer value from a table lookup matching the 3-character “AAA” type identifier component of the *AAA\_N\_<meta|std|change>.json* Data Product file; value is table look-up) |
| **19-22**  (4 bits) | | *dp\_instance* | [ 0 - 15] | **Data Product “Instance”**  (equivalent to the SDK’s “instance;” an integer value matching the N-digit component of the *AAA\_N\_<meta|std|change>.json* Data Product file) |
| **23-29**  (7 bits) | | *dp\_index* | [ 0 - 127 ]  ***mod 128*** | **Data Product “Index”**  (a Bot-specific, sequential indexed count, incremented by 1, representing the number of uplink transmissions for this *dp\_type* and *dp\_instance* - both ‘std’ and ‘change’; cloud must anticipate a mod 128 rollover). |
| **30**  (1 bit) | | *data\_kind* | [ 0 | 1 ] | **Data “Kind” Flag**  (indicates whether this Data Product segment contains a “Standard” or a “Change’ component; if “0,” then ‘standard’ data info is included; if “1,” then ‘change’ data info is included) |
| **31**  (1 bit) | | *data\_change* | [ 0 | 1 ] | **Data “Change” Flag**  (if *data\_kind* above is “0,” this field has no meaning and should be ignored; if *data\_kind* above is “1,” a “0” in this field indicates “NC,” or “No Change” from previous reading, otherwise, a “1” in this field indicate a “change’ from the previous reading follows. |

Table 9: First 4 bytes (32-bits) for ALL Uplink Message "Data Segments"

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Bot Uplink Message**  **ALL Data Segments**  **Bit Map (Pt. 2)** | | | **Element Name** | **Range/Format** | **Description/Notes** |
| **Bytes 4-11 (64-bits)** | **0-11**  (12 bits) | | *status\_id* | [ 0 - 4095 ]  ***mod 2\*\*12*** | **Status Record ID**  (the ‘primary key’ or ‘rowid’ that uniquely identifies this Status Record in the ‘status’ Table of the Float’s embedded database; this built-in SQLite field can range from 0 - 264-1 on the Float; must anticipate rollover) |
| **12-27**  (16 bits) | | *heading* | [ 0 - 36,000 ]  ***centidegrees*** | **Heading**  (current reading; magnetic north; division by 100 will yield ‘heading’ in degrees as a float) |
| **26-31**  (6 bits) | **28**  (1 bit) | *pitch\_sign* | [ 0 | 1 ] | **Pitch Sign Bit**  (sign bit for the current pitch angle; 0=positive; 1=negative) |
| **29-33**  (5 bits) | *pitch* | [ 0 - 31 ] | **Pitch**  (current pitch angle; represented in degrees; must apply *pitch\_sign* above) |
| **32-37**  (6 bits) | **34**  (1 bit) | *roll\_sign* | [ 0 | 1 ] | **Roll Sign Bit**  (sign bit for the current roll angle; 0=positive; 1=negative) |
| **35-39**  (5 bits) | *roll* | [ 0 - 31 ] | **Roll**  (current roll angle; represented in degrees; must apply *roll\_sign* above) |
| **40-43**  (4 bits) | | *channels* | [ 0 - 15 ] | **Channels**  (data channels for this Data Product) |
| **44-47**  (4 bits) | | *dt* | [ 0 | 1| 2 | 3 ] | **Data Type**  (0=uint8; 1=int32; 2=float32 ; 3=Reserved, TBD) |
| **48-59**  (12 bits) | | *size* | [ 0 - 4095 ] | **Data Size**  (the total number of ‘standard’ or ‘change’ data products of ***dt*** Data Types) |
| **60-63**  (4 bits) | | *-* | - | **Reserved** |

Table 10: Next 8 bytes (64-bits) for ALL Uplink Message "Data Segments”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bot Uplink Message STD Data Segment**  **Bit Map (Pt. 3)** | | **Element Name** | **Range/Format** | **Description/Notes** |
| **Bytes 12-15 (32-bits)** | **0-9**  (10 bits) | *samples* | [ 0 - 1023 ] | **Samples**  (number of ‘samples’ for this Data Product) |
| **10-20**  (11 bits) | *rows* | [ 0 - 2047 ] | **Rows**  (number of ‘rows’ in a ‘sample’ for this Data Product) |
| **21-31**  (11 bits) | *cols* | [ 0 - 2047 ] | **Columns**  (number of ‘columns’ in a ‘sample’ for this Data Product) |
|  | **0 🡪 n**  **(values repeat** *size* **times)** | *data* | byte array | **Data**  (a data stream of “***size\*1***” bytes when ***dt=0*** OR “***size\*4***” bytes when ***dt=1|2***; the stream starts at byte 16 in a ‘std’ Data Segment) |

Table 11: Bytes 12+ in Data Segment when "std" Data Product Information is included in the Segment

### Message Signing

*TBD*

### Message Checksum

*TBD*

### Message Compression

*TBD*

### Message Encryption

*TBD*

### Housekeeping

*TBD*

## Message Delivery (Communications Manager: *botcomm.py*)

### Introduction

Bot-Comm is a class library of the NEPI-Bot Application, used by the Bot-Recv and Bot-Sendsubsystems, and is comprised of two functional components: a convenient API and a set of supported communications drivers behind the interface.

The API component is designed to be a protocol-agnostic interface that provides a uniform set of function calls, buffering all NEPI-Bot subsystems from the intricacies of communications handshaking, packet sizing, packet ordering and reconstruction, message acknowledgements, error correction, etc. The API provides the Application with a simple way to: 1) establish connections to “the Cloud,” 2) receive messages from and send messages to “the Cloud,” 3) manage protocol-related message packetization, 4) manage message acknowledgement functionality, and 5) flush and terminate unneeded connections.

The primary *(i.e.*, production default) transmission protocol is Iridium, using the Iridium Short Burst Device on the Float and communicating with the Iridium Subscriber Unit in “the Cloud.” Other communication protocols, like Ethernet, RS-232, Wi-Fi, etc., provide alternative delivery methods and useful testing capabilities.

### Communications Connection

The Bot-Comm API provides for creating a new communication object in the following manner:

import bot-conn

conn = new BotConn(string *Conns.IRID | Conns.ETH | Conns.RS232 | Conns.WIFI*)

Because the programming language for Bot-Comm is likely Python 2.7, the alternative methodology for providing enum-like capabilities to the application might be to use a simple class to implement that functionality. In the bot-comm.py library, use a class similar to the following:

class Conns:

IRID, ETH, RS232, WIFI = range(1,5)

conn\_default = Conns.IRID # Set “Factory Default” value to Iridium

There are two possibilities for establishing the actual communications connection: 1) as part of the class constructor itself, the connection could be established or 2) for added flexibility, create a function that can be called as-needed by the application or subsystem involved (*e.g.*, Bot-Recv, Bot-Send, etc.), such as:

conn.open()

Depending on the communication protocol selected, it is possible that the actual connection might be established differently. For example, one protocol might establish its connection in the class constructor with a subsequent call to open() doing nothing. Using another protocol, new might simply return the class object and establish the connection later with the call to open().

### Inbound Messaging

*in progress with Kevin/Jacob …*

### Outbound Messaging

*in progress with Kevin/Jacob …*

### Message Acknowledgement

*in progress with Kevin/Jacob …*

### Communications Termination

Because communication connections may be handled differently depending on the protocol (e.g., socket connections in Python typically employ shutdown() followed by close()), Bot-Comm implements a 2-step approach to connection closure:

conn.close()

conn.destroy()

Although they can be called by an application or subsystem at different times, close() and destroy() should be called in that order. The call to close() closes (i.e. disconnects) the connection but leaves any connection object(s) intact. The call to destroy() should destroy any connection objects, thus requiring a new class instance.

### Housekeeping

*TBD*