**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 |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

Contents

[Purpose 8](#_Toc536781638)

[Reference Documents 8](#_Toc536781639)

[Program Architecture 9](#_Toc536781640)

[The Ocean Float Program 9](#_Toc536781641)

[The NumSDK Application 9](#_Toc536781642)

[The NEPI-Bot Application 9](#_Toc536781643)

[The NEPI-Portal Application 9](#_Toc536781644)

[NEPI-Bot General Architecture 10](#_Toc536781645)

[Introduction 10](#_Toc536781646)

[NEPI-Bot Application Design 11](#_Toc536781647)

[Bot-Recv Subsystem (*bot-recv.py*) 11](#_Toc536781648)

[Bot-Send Subsystem (*bot-send.py*) 11](#_Toc536781649)

[Bot-DB Class Library (*bot-db.py*) 12](#_Toc536781650)

[Bot-PIPO Class Library (*bot-PIPO.py*) 12](#_Toc536781651)

[Bot-Mess Class Library (*bot-mess.py*) 12](#_Toc536781652)

[Bot-Help Library (*bot-help.py*) 12](#_Toc536781653)

[Bot-Comm Class Library (*bot-comm.py*) 12](#_Toc536781654)

[NEPI-Bot Configuration File 13](#_Toc536781655)

[Bot-Recv Subsystem (the NEPI-Bot Downlink Manager: *bot-recv.py*) 16](#_Toc536781656)

[Introduction 16](#_Toc536781657)

[Assumptions 16](#_Toc536781658)

[Subsystem Configuration 16](#_Toc536781659)

[Comm Link Instantiation 16](#_Toc536781660)

[Inbound NEPI-Bot Message Consumption 16](#_Toc536781661)

[Inbound NumSDK Message Distribution 16](#_Toc536781662)

[Housekeeping and Termination 16](#_Toc536781663)

[Bot-Send Subsystem (the NEPI-Bot Uplink Manager: *bot-send.py*) 17](#_Toc536781664)

[Introduction 17](#_Toc536781665)

[Assumptions 17](#_Toc536781666)

[Subsystem Configuration 17](#_Toc536781667)

[Status and Data Retrieval 18](#_Toc536781668)

[Persistent Storage (Database Manager: *bot-db.py*) 19](#_Toc536781669)

[Introduction 19](#_Toc536781670)

[Database Instantiation 19](#_Toc536781671)

[Status Table 19](#_Toc536781672)

[Data Product Tables 21](#_Toc536781673)

[Housekeeping 22](#_Toc536781674)

[Data Product Prioritization (PIPO Manager: *bot-pipo.py*) 22](#_Toc536781675)

[Introduction 22](#_Toc536781676)

[Status and Data Product Consumption 22](#_Toc536781677)

[Data Product Prioritization 22](#_Toc536781678)

[Upload Message Buffer Creation 23](#_Toc536781679)

[Data Product Selection 23](#_Toc536781680)

[Message Packetization 23](#_Toc536781681)

[Data Product Purging 24](#_Toc536781682)

[Housekeeping 24](#_Toc536781683)

[Upload Message Buffer Packaging (Message Manager: *bot-proc.py*) 24](#_Toc536781684)

[Introduction 24](#_Toc536781685)

[Message Creation 24](#_Toc536781686)

[Request Messages 24](#_Toc536781687)

[Message Bit-Packing 24](#_Toc536781688)

[Message Signing 30](#_Toc536781689)

[Message Checksum 31](#_Toc536781690)

[Message Compression 31](#_Toc536781691)

[Message Encryption 31](#_Toc536781692)

[Housekeeping 31](#_Toc536781693)

[Message Delivery (Communications Manager: *bot-comm.py*) 31](#_Toc536781694)

[Introduction 31](#_Toc536781695)

[Communications Connection 31](#_Toc536781696)

[Inbound Messaging 32](#_Toc536781697)

[Outbound Messaging 32](#_Toc536781698)

[Message Acknowledgement 32](#_Toc536781699)

[Communications Termination 32](#_Toc536781700)

[Housekeeping 32](#_Toc536781701)

**Table of Figures**

[Figure 1: The NEPI-Bot Application with Supporting Subsystems 10](#_Toc536781702)

[Figure 2: Bot-Send List of “Data Folders” in the “Data Directory” (created by NumSDK) 18](#_Toc536781703)

**Table of Tables**

[Table 1: The NEPI-Bot Configuration File (JSON Formatted) 15](#_Toc536781704)

[Table 2: Status File JSON Mappings to Python Datatypes and SQLite Type Affinity 21](#_Toc536781705)

[Table 3: Data File JSON Mappings to Python Datatypes and SQLite Type Affinity 21](#_Toc536781706)

[Table 4: NumSDK Status Record to Cloud Message Bit-Packing Map 30](#_Toc536781707)

**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 the Iridium SDB, Ethernet, RS-232, Wi-Fi, and/or other technologies.

# 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 B; 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 are the field test engineers, product and intelligent peripheral developers.

# Program Architecture

## The Ocean Float Program

Numurus’ **Ocean Float Program** is comprised of: 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 its 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 on those targeted to 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 the wen-user’s interface and supports 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).

Figure : The NEPI-Bot Application with Supporting Subsystems

**Cloud**

**Float**

**API**

**NEPI-Bot**

**Bot-DB**

(SQLite)

**Bot-Send**

**Status**

**& Data**

**Files**

**Config**

**Files**

**Bot-Recv**

**Bot-Comm**

**Other Driver**

**Wi-Fi Driver**

**RS-232 Driver**

**Ethernet Driver**

**Iridium Driver**

**NEPI**

SDK

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 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 large, check-summed, configurable message buffers for delivery to subscribing Applications, external to the Float itself.

Finally, the NEPI-Bot subsystem 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 gathering-prioritization-compression-transmission cycle and prior to its inevitable suspension/termination, NEPI-Bot performs various housekeeping duties to insure data integrity and prevent the 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-DB, Bot-PIPO, Bot-Mess, 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 subsystem 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 information, recently sampled by the Numurus SDK, for subsequent processing. The primary responsibility of Bot-Send is to: 1) prioritize both new and old (unsent from previous samplings) Status and Data Product records, 2) cleverly package them using bit-compaction and lossless compression, 3) sign them and provide a checksum, 4) encrypt them when necessary, and 4) transmit them to “the Cloud.”

## Bot-Recv Subsystem (*bot-recv.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 messages may be targeted for NEPI-Bot or Numurus SDK.

In the former case, Bot-Recv decompresses, verifies, unpacks, and evaluates the inbound messages and 1) acts on the information immediately (*i.e.*, Bot-targeted messages) and/or 2) stores the information for use in subsequent wake-cycles (*i.e.*, NumSDK-targeted messages). For those alarm, control, and configuration messages targeted for the Numurus SDK, Bot-Recv creates the necessary JSON-formatted API files (per formatting described in the NumSDK - NEPI-Bot ICD” document). After taking all necessary actions on the inbound messages, Bot-Recv deletes the current cycle’s received messages and performs other necessary housekeeping activities before suspending/terminating.

## Bot-Send Subsystem (*bot-send.py*)

At an appropriate “later time” in a Float wake-cycle (see the “NumSDK - NEPI-Bot ICD” document), the Bot-Send Application is “forked” by the Numurus SDK. In summary, Bot-Send utilizes the Bot-DB, Bot-PIPO, Bot-Mess, Bot-Help, and Bot-Comm class libraries, and other feature-rich library components to: 1) instantiate with the current configuration file settings, 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, 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, 8) establish a connection with and transmit to “the Cloud,” and 9) perform necessary housekeeping before suspension/termination.

## Bot-DB Class Library (*bot-db.py*)

Bot-DB is a class library of NEPI-Bot Application, used by both the Bot-Recv and Bot-Sendsubsystems, providing 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 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 deletion).

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

Bot-PIPO is a class library of NEPI-Bot Application, used primarily by the Bot-Send subsystem, and acts as the Priority-In Priority-Out (PIPO) Manager. Bot-PIPO’s job is to take the complete list of status and data 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 and general housekeeping.

## Bot-Mess Class Library (*bot-mess.py*)

Bot-Mess is a class library of NEPI-Bot Application, used by both the Bot-Recv subsystem 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-Mess 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-Help Library (*bot-help.py*)

Bot-Help is a 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, etc.

## Bot-Comm Class Library (*bot-comm.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) 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. 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.

# NEPI-Bot Configuration File

The NEPI-Bot Configuration File provides instantiation and configuration information to the various NEPI-Bot Applications, subsystems, and 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).

All subsystems and components comprising the NEPI-Bot Application assume the existence of the NEPI-Bot Configuration File. In the absence of this configuration file, a set of built-in Factory Default values are implemented to enable to NEPI-Bot Application to function, 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.

|  |  |  |  |
| --- | --- | --- | --- |
| **Element Name** | **JSON Type** | **Range/Format** | **Description/Notes** |
| *dbg\_mode* | integer | **[0-3]**  Factory Default: ***0*** | **Debug Mode**  (controls the nature of internal coding, including Logging. ‘0’ is OFF, debug level is 1-3; the higher the debug level, the more detailed the report) |
| *log\_mode* | integer | **[0-3]**  Factory Default: ***0*** | **Logging Mode**  (controls the nature of internal coding, including Logging. ‘0’ is OFF, debug level is 1-3; the higher the debug level, the more detailed the report) |
| *host* | string | **Class A, B, or C IP Addresses**  Default: ***10.0.0.116*** | **Host IP Address**  (the IP address used by NEPI Bot for TCP/IP connections. Using *10.0.0.116* for ALPHA or local testing) |
| *port* | integer | **[49152 - 65535]**  Default: ***7770*** | **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) |
| *protocol* | integer | **[0-4]**  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) |
| *packet\_size* | integer | **[1024-n]**  Default: ***1024*** | **Protocol Packet Size**  (used in conjunction with ‘protocol,’ Packet Size represents the desired or required packet size for the selected protocol) |
| *sys\_status\_file* | string | **<*file\_name*>.json**  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 info). |
| *data\_dir* | string | **<*relative\_path*>**  Factory Default: ***data/*** | **Data Directory**  (location of the Float’s Data Directory, relative to the Float’s home directory, which on the Float device is: */home/nepi-user/*) |
| *db\_dir* | string | **<*relative\_path*>**  Factory Default: ***db/*** | **Database Directory**  (location of the Float’s Data Directory, relative to the Float’s home directory, which on the Float device is: */home/nepi-user/*) |
| *db\_name* | string | **<*file\_name*>.<*ext*>**  Factory Default: ***float.db*** | **Database File Name**  (the “ext” is optional but, for SQLite, the extension is traditionally “.db”) |
| *pipo\_scor\_wt* | float | **[0.0 - 1.0]**  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) 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: *bot-pipo.py*) section of this document) |
| *pipo\_qual\_wt* | float | **[0.0 - 1.0]**  Default: **0.5** |
| *pipo\_size\_wt* | float | **[0.0 - 1.0]**  Default: **0.5** |
| *pipo\_time\_wt* | float | **[0.0 - 1.0]**  Default: **0.5** |
| *pipo\_trig\_wt* | float | **[0.0 - 1.0]**  Default: **0.5** |
| *purge\_rating* | float | **[0.0500 - 0.9000]**  Default: ***0.1000*** | **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]**  Default: ***65536*** | **Maximum Message Size**  (maximum size in characters of the outbound Cloud message after bit-packing, compression, signing, checksum, encryption, etc.) |

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

# Bot-Recv Subsystem (the NEPI-Bot Downlink Manager: *bot-recv.py*)

## Introduction

## Assumptions

## Subsystem Configuration

When awakened by the Numurus SDK at appropriate, configurable times, Bot-Send consumes the “NEPI-Bot Configuration File. For details regarding the format, contents, defaults, and uses of the NEPI-Bot Application’s parameter and attribute settings, see the “NEPI-Bot Configuration File” section earlier in this document.

## Comm Link Instantiation

## Inbound NEPI-Bot Message Consumption

## Inbound NumSDK Message Distribution

## Housekeeping and Termination

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

## Introduction

Along with Bot-Recv, Bot-Send is one of two primary subsystems in the NEPI-Bot Application and it is certainly the more complex. 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-DB, Bot-PIPO, Bot-Mess, 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 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 component, 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, relation database.
3. *Prioritizing Data Products* by using the Bot-PIPO library component to evaluate each new Data Product record according to a prescribed formula (see: PIPO management, elsewhere in this document) and, when necessary, re-evaluating previously-prioritized Data Products 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-Mess, 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 (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 Cloud,” transmitting the entire message buffer, 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 for the official “Data Directory” (as described in the “NumSDK - NEPI-Bot ICD”), and execute other necessary clean-up requirements.

## Assumptions

The Bot-Send subsystem is awakened 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.”

*more assumptions coming …*

## Subsystem Configuration

When awakened by the Numurus SDK at appropriate, configurable times, Bot-Send consumes the “NEPI-Bot Configuration File. For details regarding the format, contents, defaults, and uses of the NEPI-Bot Application’s parameter and attribute settings, see the “NEPI-Bot Configuration File” section earlier in this document.

## Status and Data Retrieval

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

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 that 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 2* 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**

Figure : 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.

Each Data Folder contains *n*-number of JSON-formatted Data Product files, along with one, and only one, JSON-formatted “Status” file (using a configurable filename as contained in the Bot Configuration File; see above) that represents system status information. common to all Data Product files in that folder, at the time of the sampling.

The Data Product files are generally split into a meta-data component (*<Node\_ID>\_data.json>*) and, when needed, an associated binary data component (*<Node\_ID>\_data.ext>*), both residing in the Data Folder and associated with the time of Data Product collection/processing (node-type dependent). The binary component Data Product file shares the same name with its metadata file but uses an extension commensurate with the Node Type it represents. See the “NumSDK - NEPI-Bot ICD” for a detailed description of the various Data Product files, extensions, and their contents.

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 are described in detail in the “NumSDK - NEPI-Bot ICD” document. Where appropriate, these file formats will be reprised in this document or expanded upon as necessary to support NEPI-Bot Application requirements.

## Persistent Storage (Database Manager: *bot-db.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 also has the ability to implement 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. ]

### Database Instantiation

The primary NEPI-Bot subsystems, Bot-Recv and Bot-Send, have the ability to check on start-up for the existence of the Float’s Database: *float.db*. If the Float’s Database does not exist, the first subsystem to notice its absence will create the embedded Database and instantiate all necessary Tables and Indexes.

### Status Table

*write-up coming …*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Column Name** | **JSON Type** | **Python**  **Type** | **SQLite**  **Affinity** | **Range/Format** | **Description/Notes** |
| *row\_id* | - |  | INTEGER | [ 1 - inf ] | **Status Record Index**  (*implicit* autoincrement column; 64-bit signed integer) |
| *reserved* | - |  |  |  |  |
| *reserved* | - |  |  |  |  |
| *timestamp* | number | float | REAL | [ (0.0 - 232) ] | **Unix Epoch Time**  (millisecond resolution) |
| *serial\_num* | string | string | STRING | TBD |  |
| *sw\_rev* | string | string | STRING | TBD | **Software Revision**  (specifies overall s/w revision for the system. Independent of config index values) |
| *navsat\_fix\_time* | number | integer | INTEGER | [ 0 - 232 ] | **Unix Epoch Time**  (last satellite fix) |
| *latitude* | number | float | REAL | [ -90.0 - 90.0 ] | **Latitude Position**  (positive is north of equator) |
| *longitude* | number | float | REAL | [ -180.0 - 180.0 ] | **Longitude Position**  (positive is east of prime meridian) |
| *heading* | number | float | REAL | [ 0.0 - 360.0 ] | **Float Heading**  (magnetic north) |
| *batt\_charge* | number | float | REAL | [ 0.0 -100.0 ) | **Battery Charge**  (percentage charge remaining) |
| *bus\_voltage* | number | float | REAL | TBD | **Main Bus Voltage** |
| *temperature* | number | float | REAL | TBD | **Main Temperature** |
| *trig\_wake\_count* | number | integer | INTEGER | [ 0 - 10000 ] | **Trigger Wake Count**  (number of times Zynq has awoken due to sensor detection triggers) |
| *wake\_event\_type* | number | integer | INTEGER | 0: alarm, 1: trigger | **Wake Event Type**  (specifies the event that initiated the data collection) |
| *wake\_event\_id* | number | TBD | TBD | *task\_id* for alarms, *smarttrigger\_id* for trigger | **Wake Event ID**  (interpretation depends on value of the *wake\_event\_type*) |
| *task\_index* | number | integer | INTEGER | [ 0 - 256 ] | **Task Index**  (most recent task schedule update index) |
| *trig\_cfg\_index* | number | integer | INTEGER | [ 0 - 256 ] | **Trigger Config Index**  (most recent SmartTrigger configuration index) |
| *rule\_cfg\_index* | number | integer | INTEGER | [ 0 - TBD ] | **Rule Config Index**  (most recent SmartTrigger rule modification index) |
| *sensor\_cfg\_index* | number | integer | INTEGER | [ 0 - TBD ] | **Sensor Config Index**  (most recent sensor configuration index) |
| *node\_cfg\_index* | number | integer | INTEGER | [ 0 - TBD ] | **Node Config Index**  (most recent Node configuration index) |
| *state\_flags* | number | integer | INTEGER | 32-bit mask. Contents TBD. | State Flag Mask  (*e.g.*, temperature and storage warnings, drag line state, etc.) |

Table : Status File JSON Mappings to Python Datatypes and SQLite Type Affinity

### Data Product Tables

The JSON-formatted Data Product files are generally split into a metadata component and an associated binary data component, both residing (with the same timestamp) in the Data Folder. The is associated with the time of data collection/processing (node-type dependent).

*write-up coming …*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Column Name** | **JSON Type** | **Python**  **Type** | **SQLite**  **Affinity** | **Range/Format** | **Description/Notes** |
| *row\_id* | - | integer | INTEGER | [ 1 - inf ] | **Status Record Index**  (*implicit* autoincrement column; 64-bit signed integer) |
| *reserved* | - |  |  |  |  |
| *reserved* | - |  |  |  |  |
| *status\_record* | - | integer | INTEGER | [ 1 - inf ] | **Status Record (foreign key)**  (a pointer to the ‘*row\_id*’ associated Status Record in the Status Table) |
| *heading* | number | float | REAL | [ 0.0 - 360.0 ] | **Float Heading**  (the current reading; magnetic north) |
| *quality* | number | integer | INTEGER | [ 0 - 100 ] | **Quality Indicator**  (a node-dependent indicator of quality; used in prioritizing data) |
| *bin\_file* | string | string | TEXT | <file\_name>.<ext> | **Binary File**  (same name as this file, but with different, node-dependent extension; NULL if no binary data for this node type) |
| *data* | TBD | TBD | TBD | TBD | **Node Data**  (nodes may embed data directly in this file; see TBD for node-dependent column details; NULL if no embedded data for this node type) |

Table : Data File JSON Mappings to Python Datatypes and SQLite Type Affinity

### Housekeeping

## Data Product Prioritization (PIPO Manager: *bot-pipo.py*)

### Introduction

The Priority-In Priority-Out (PIPO) Manager’s job is to take a folder of Data Products as delivered by the Numurus SDK, store all status and data records in the Float’s embedded database, prioritize the data record (*i.e.*, calculate its PIPO Score), and, with aging, purge status and data information that falls 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: *bot-db.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 the complete list of Data Products (*i.e.*, all status and data records captured and stored by the Numurus SDK since NEPI-Bot’s last “wake-up”) and compute (or, when necessary, re-compute previously-stored records) a prioritization value based on a prescribed, configurable formula, employing the following weighting characteristics:

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

In the above formula, A, B, C ,D, & E are Cloud-configurable weight factors found in the Bot Configuration File, where A=*pipo\_scor\_wt*, B=*pipo\_qual\_wt*, C=*pipo\_size\_wt*, D=*pipo\_trig\_wt*, and E=*pipo\_time\_wt*. All weight values range from [0.0 - 1.0] and the four weight values used to calculate the formula’s “numerator” *(i.e.*, A, B, C, and D) are normalized to a 4d unit vector, that is, having a magnitude of 1 (within a small epsilon).

The weighting factor’s unit vector is calculated according to the following formula:

unit vector = vector / magnitude of the vector

or, in mathematical terms,



*detailed implementation formula and examples coming …*

The E weighting factor must be ≥0 to avoid division by 0, but otherwise unrestricted. The Data Product’s “Age” represents a fixed time unit (*i.e.*, days, hours, min., sec., etc.). Technically, it doesn't matter which unit is used (since the E weighting factor will scale it), but the “hours” unit has been selected as being a reasonable representation within the scope of a limit Float lifespan. The addition of “1” to the Age avoids division by 0 when the “Age” of the Data Record is sufficiently short to round possibly to 0.

The above constraints assure that 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 in one or more previous wake-up cycles). Calculated “Numerator” values for previously-evaluated Data Records require recalculation only in the event of a Cloud 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.

### Housekeeping

*TBD*

## Upload Message Buffer Packaging (Message Manager: *bot-proc.py*)

### Introduction

### Message Creation

### Request Messages

*currently in progress with Josh/Alex …*

### Message Bit-Packing

*currently in progress with Josh/Alex and Clint/Jacob …*

*This table needs Description and Notes combined into one, coherent statement in Description column and finalization details regarding the “required bits” and “message bits” information.*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Element Name** | **JSON Type** | **Range/Format** | **Description/Notes** | **Req**  **Bits** | **Msg**  **Bits** | **Notes** |
| *timestamp* | number | Unix Epoch with millisecond resolution  In the JSON file, is the *number* a string representation of a float (w/3 decimal precision) or an int (that is, x 1000; with no decimal point)? The first, a float. | Represents the wakeup time (when status initially collected) and should match the containing directory’s name. | 42 | 0-41 | Unix Epoch requires 32 bits (lower order 31 bits of UNIX timestamp are currently in play). mSec (x 1000) requires additional 10 bits. But we should consider alternative schemes, like converting this to a rolling Epoch based on a timestamp that cloud and float can agree upon, maybe last transmitted status timestamp serves as the zero-point for the next one. Moreover, can truncate/round off the milliseconds. This could bring us closer to 15-16 bits (assuming two connections per day, rolling the clock over if we miss one). |
| *serial\_num* | string | TBD - Does this assume printable ASCII char set (Unicode 32-126)? |  | TBD | TBD | Will likely be converted to char[] for inclusion in message stream. Probably want to save the character bytes and use a numeric identifier. We could even implement that as a Numurus standard at this point, disallowing non-numeric characters in serial numbers. |
| *sw\_rev* | string | TBD - Does this assume printable ASCII char set (Unicode 32-126)? | Specifies overall sw rev for the system. Independent of configuration index values. | TBD | TBD | Will likely be converted to char[] for inclusion in message stream. If this is included in the message stream at all, it should be converted to a small number of numeric bits (12-15, broken into semantic versioning elements). |
| *navsat\_fix\_time* | number | [0.0, Unix Epoch Max] | Unix Epoch Time for last satellite fix | TBD | TBD | Like the *timestamp* Element above, does this imply mSec resolution? And how is it represented per the discussion above? For message transmission purposes, it should be treated just as the regular timestamp above |
| *latitude* | number | [- 90.0,90.0] How many decimal places [see discussion on right]? | Positive is north of equator | TBD | TBD | What is the limit on decimals decimal we plan to report? The 5th decimal place is accurate to 1.1 m and accuracy to this level with commercial GPS units can only be achieved with some differential correction. Commonly, Latitude and Longitude are expressed using precision to 6 decimal places, accurate to 0.11 m, that is achieved with painstaking measures with differentially-corrected GPS. The 7th decimal place gets us to 11 mm and is at the limit of what GPS-based techniques can achieve. This can be a configurable item, but it will change the bit positioning of the uploaded message contents. Alternatively, we can just use 5, 6, or 7 decimal places as the limit and leave it at that. With *heading*, if integer precision is acceptable, bit packing can be reduced to 9 bits + 1 sign bit (total of 10 bits) In some applications that we target, position is determined by non-GPS sensors with higher accuracy (at least in a relative sense), so we should probably assume millimeter precision is required. But, does it matter? How would you encode this data into a smaller-than-single-precision-float size element? I suppose you could represent it as (integer) millidegrees, in which case you might need just 29 bits for 360 degrees, but the savings are pretty minor (however – I’m rusty on floats, and it may be that you pay a lot of loss-of-precision penalty down this many decimal points). |
| *longitude* | number | (-180.0,180.0] How many decimal places [see discussion on right]? | Positive is east of prime meridian | TBD | TBD |
| *heading* | number | [0.0,360.0) How many decimal places [see discussion on right]? | Magnetic North | TBD | TBD |
| *batt\_charge* | number | (0.0,100.0] How many decimal places [see discussion on right]? | Charge remaining (percentage) |  | TBD | If integer precision is acceptable for reporting, bit packing can be reduced to 7 bits. Yes, I think that’s fine. |
| *bus\_voltage* | number | TBD | Main bus voltage | TBD | TBD | Similar discussion to *batt\_charge* |
| *temperature* | number | TBD | Main temperature | TBD | TBD | Similar discussion to *batt\_charge* |
| *trig\_wake\_count* | number | [0,inf) Need a realistic max value associated with “infinity.” Email discussions suggested < 10,000 is reasonable for 30 “wake-ups” a day for a year (i.e., generous lifetime of a Float). | Number of times Zynq has awoken due to sensor detection triggers | TBD | TBD | Clint and John can discuss other options here (such as wake-up count since last Iridium connection). This approach will require on-Float retention of previous wake-up count values. |
| *wake\_event\_type* | number | 0: alarm, 1: trigger | Specifies the event (trigger or alarm) that initiated the data collection. | 1 | TBD | Any possibility this could be moved to a the *state\_flag* below? Yes, though I want to truncate the state\_flag to the actual number of bits we need (see below) – In that case, there is no net gain to moving this. |
| *wake\_event\_id* | number | *task\_id* for alarms, *smarttrigger\_id* for trigger | Interpretation depends on the value of the *wake\_event\_type* field | TBD | TBD | smarttrigger\_id appears to be in the range of 0-7 currently, with options for more. Is there a realistic max value? Let’s cap it at 256 (consistent with other caps below). The task\_id appears to be 0|1|2|3|1000. Is it realistic to replace 1000 with either 9 or 99? Yes. |
| *task\_index* | number | [0,inf) Need a realistic max value associated with “infinity.” Let’s say 256. | Most recent task schedule update index. | TBD | TBD | Similar to earlier discussions, a reasonable estimate of the possible task schedule and configuration index records is valuable. If <= 256 records are anticipated for the upper limit of stored records (for any given type), these Element Values can be uploaded using just 8 bits. Different upper limits can be accommodated. 256 for everybody! |
| *trig\_cfg\_index* | number | [0,inf) Need a realistic max value associated with “infinity.” 256. | Most recent SmartTrigger configuration index | TBD | TBD |
| *rule\_cfg\_index* | number | [0,inf) Need a realistic max value associated with “infinity.” | Most recent SmartTrigger rule modification index | TBD | TBD |
| *sensor\_cfg\_index* | number | [0,inf) Need a realistic max value associated with “infinity.” | Most recent sensor configuration index | TBD | TBD |
| *node\_cfg\_index* | number | [0,inf) Need a realistic max value associated with “infinity.” | Most recent Node configuration index | TBD | TBD |
| *state\_flags* | number | 32-bit mask. Contents TBD. I was a little cavalier here. I will generate an actual state table and we can cut this down to a more appropriate number of bits. | E.g., temperature and storage warnings, drag line state | 32 | TBD | Stored as-is in 32-bit (4 char). |

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

### Message Signing

*TBD*

### Message Checksum

*TBD*

### Message Compression

*TBD*

### Message Encryption

*TBD*

### Housekeeping

*TBD*

## Message Delivery (Communications Manager: *bot-comm.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*