Tim -

RS-232 (2 UARTs on the Zynq), Ethernet (Std 802.3), and Iridium SBD (presumably, an RS-232 interface), as possible choices for in-boarding command and configuration messages and out-boarding the Float’s data streams.  Each of these communication methods requires some kind of software driver with an API that could be used by NEPI-Bot subsystem to send messages to and receive messages from "the Cloud" (which is currently understood be a "transparent” (or, pass-through) pipeline to the NEPI Portal).

**“Surface”**

**Float**

**6**

**5**

**9**

**8**

**7**

**4**

**3**

**2**

**1**

**“CLOUD”**

Kafka

Laptop

Wire

USB

Etc.

**NEPI**

Portal

***Figure 1. End-to-end depiction of the Numurus’ Ocean Float Program.***

In order to capture a sense of the communications needs for completing end-to-end transmission of 1) command, control, and configuration messages from “the Cloud” to the Float and 2) various data product streams from the Float to “the Cloud,” the following nine subsystems must be understood independently and in relation to each other.

1. The “**Numurus SDK**” Subsystem (Josh/Alex). Essentially, the SDK samples various status and data points available on the Float and, thus, represents the start of the Float-to-Cloud transmission process. It captures the data points and stores them on the SD card in JSON file format.
2. The “**Numurus SDK – NEPI-Bot API**” (Josh/Alex). These are JSON-formatted files containing status and data points. The files are written by the Numurus SDK and read by the NEPI-Bot “data cruncher” using an “advisory” Linux file locking mechanism to prevent collision problems.
3. The “**NEPI-Bot**” Subsystem (John). NEPI-Bot retrieve the status and data files (created by the Numurus SDK), manipulates the data according to a PIPO process (a priority evaluation), compacts/compresses the data, formats a configurable stream of status and data information, and sends the information to “the Cloud.”
4. The “**NEPI-bot -Cloud API**” (?). This is a communications API that provides an agnostic programming interface for communicating to and receiving from “the Cloud” without consideration for the actual hardware interface or implementation detail (e.g., Iridium SBD, Wi-Fi, Ethernet, RS-232, etc.).
5. The “**Float-Level Communication Drivers and API**” (?). Driver-level software to manage the various hardware communications ports available on the Zynq-7000 Soc. These can include the Iridium SDB, Wi-Fi, Ethernet, RS-232. etc.
6. The “**Cloud-Level Interface Drivers and APIs**” (Clint + DARPA; and ?). Additional (non-Float) drivers are required for external platforms to send to and receive from the Float device. Currently, an Iridium Receiver, Event Processor (Kafka), and NEPI-Portal/Storage are under development for the “surface-side” of the Ocean Float Program. In order to “connect to” the Float via other access points (Wi-Fi, Ethernet, RS-232), the acquisition of appropriate applications, APIs, and drivers, are required.
7. The “**Cloud**.” Currently, “the Cloud” is comprised of a Navy/DARPA-driven development effort that includes the Iridium Satellite System, an Iridium Receiver, a pub/sub application-level “data distribution service” (based on Apache Kafka), and a Numurus initiative (the NEPI Portal, NEPI Storage, and NEPI API). The “Cloud,” however, can be a laptop, a USB drive, or other platform that is able to communicate via Wi-Fi, Ethernet, RS-232, etc. It can be any device capable of communicating with the Float device for the purpose of sending commands and configuration messages, along with receiving status and data streams from the Float.
8. The “**Cloud – Application API**” (Clint; ?) Currently, the NEPI API is the interface being designed to communicate with the NEPI Portal (a UI/UX front-end to the status and data streams from the Float device. For direct communication with the Float (via Wi-Fi, Ethernet, RS-232, USB file storage, etc.), this particular API requirement may simply be an end-point driver on the connected platform.
9. The “**Web Application**” (Clint). At the far end of the communications process is a typical UI/UX application, designed to allow Users to 1) send appropriate command, control, and configuration messages to the Float, 2) receive and store status and data streams from the Float, and 3) analyze, visualize, and review Float status and data. Currently, the NEPI Portal is Numurus’ universal offering for this UI/UX experience.

Figure 1 highlights a necessary subsystem (in light brown): the arbitrarily-named “NEPI-Comm API,” along with its associated software drivers, all of which supports a Float-level communications interface with “the Cloud.”

Based on the Josh/Alex/Jay/John teleconference on 1/7/2018 (2-4 pm), it was mentioned that "NAL" (?) might be considered for assistance in the development of a driver, or drivers, (with an agnostic API) to act as an interface between the NEPI-Bot and "the Cloud."

High-level requirements and statement-of-work for the “Float-Level Communication Drivers and API” subsystems (#4 and #5 above) are as follows.

1. An Iridium SBD driver for the sending/receiving of messages to/from “the Cloud.”
   1. The current Float processor is the Zynq-7000 SoC, ARM Cortex-A9, 16GB, etc.
   2. Development language is open (should be discussed).
2. At least one additional driver (RS-232, Ethernet, Wi-Fi, etc.) to provide an “alternative” link to the Float, primarily for early development and testing.
   1. RS-232 (probably through UART) or Ethernet is preferable.
   2. A Linux/Windows platform app or monitoring tool (to simulate sending/receiving to/from “the Cloud”) may be required.
3. An agnostic API, used primarily by NEPI-Bot, that allows for a uniform method of sending/receiving various types of messages (configuration, status, and data streams) to/from “the Cloud.”
   1. Development language is open (but should be discussed). Primary development language on the Float is Python 2.7.x. The API must be importable and callable from Python programs.
   2. Sample API (actual class, functions, and prototyping TBD):
      1. class cd = new *class*(mode) [ *mode* is SDB, RS232, Ethernet, Wi-Fi, etc. ]
      2. cd.open() and/or cd.init() [ depending on protocol needs]
      3. cd.send(type, message) [ *type* is status, config, data, etc.; returns *err\_code* ]
      4. cd.receive(type) [ returns *message*|*null* ]
      5. cd.close() [ returns *err\_code* ]
      6. cd.terminate(), cd.suspend(), cd.destroy() [ depending on protocol needs ]