Antaris™

SatOS™ Payload SDK Guide

Version 1.1 (2022-12-02)

Table of Contents

1. Introduction *4*

2. Revision History *4*

3. Overview *5*

3.1. Key Terms 5

3.2. System Architecture 6

4. Application Design *6*

4.1. Application Operation 7

4.2. Application Modes 7

4.3. Payload Sequences 7

4.4. Tasking & Scheduling 8

4.5. File Upload / Download 9

4.6. Application Upgrades 9

5. Using the Payload SDK *10*

5.1. Developer Deliverables 10

6. Application System Environment *10*

6.1. Configuration 10

6.2. Compute 10

6.3. Storage 11

6.4. Network 11

6.5. I/O Interface 11

7. Payload Interface *11*

7.1. Request Flow Example 12

7.2. Encryption & Authentication 12

7.3. Establishing a Connection 12

7.4. Payload Application Requests 13

7.4.1. Register Payload Application 13

7.4.1.1. Expected async response 13

7.4.2. Get Current Location 13

7.4.2.1. Expected async response 13

7.4.3. Set Payload Device Power State 14

7.4.3.1. Expected async response 14

7.4.4. Stage File Download 14

7.4.4.1. Expected async response 14
7.5. Payload Controller Requests 14
7.5.1. Start Sequence 14
7.5.1.1. Expected async response 15
7.5.2. Shutdown 15
7.5.2.1. Expected async response 15
7.5.3. Health Check 15
7.5.3.1. Expected async response 15

1. Introduction

This document describes how developers work with the Antaris platform to design, deploy and operate a payload aboard a SatOS-powered satellite.

2. Revision History

Revision	Date	Notes
1.1.0	2022-12-02	• First release

3. Overview

3.1. Key Terms

Payload Developer: Individual or group that is working with the Antaris platform to develop, deploy and operate a payload on-orbit.

Payload SDK: toolkit containing source code libraries and documentation used by Payload Developers to effectively develop a payload.

Payload Server: The physical onboard system with CPU, memory, storage and I/O connectivity for Payload Devices. It hosts Payload Applications and various system services.

Payload Controller (PC): The software stack that manages operations on the Payload Server and acts as a gateway to all services provided to Payload Applications.

Payload Interface: API enabling bi-directional communication between Payload Application and Payload Controller.

Payload Device: A physical device provided by a Payload Developer that is physically connected to the Payload Server via one of the designated interfaces (Ethernet, I2C, UART, USB, SPI, etc).

Device Driver: A general term used to describe any software supporting access to or manipulation of a Payload Device. A driver typically supports one or both of the following:

- **General Device I/O**: having generic knowledge about mechanisms to communicate with Hardware entities using a particular interface type (e.g. USB, UART etc.), or
- **Device-specific capabilities**: having knowledge of the Payload Device specifics and exposing abstracted interfaces for higher level software, while encapsulating the device details. Could be implemented as user-space software or kernel modules.

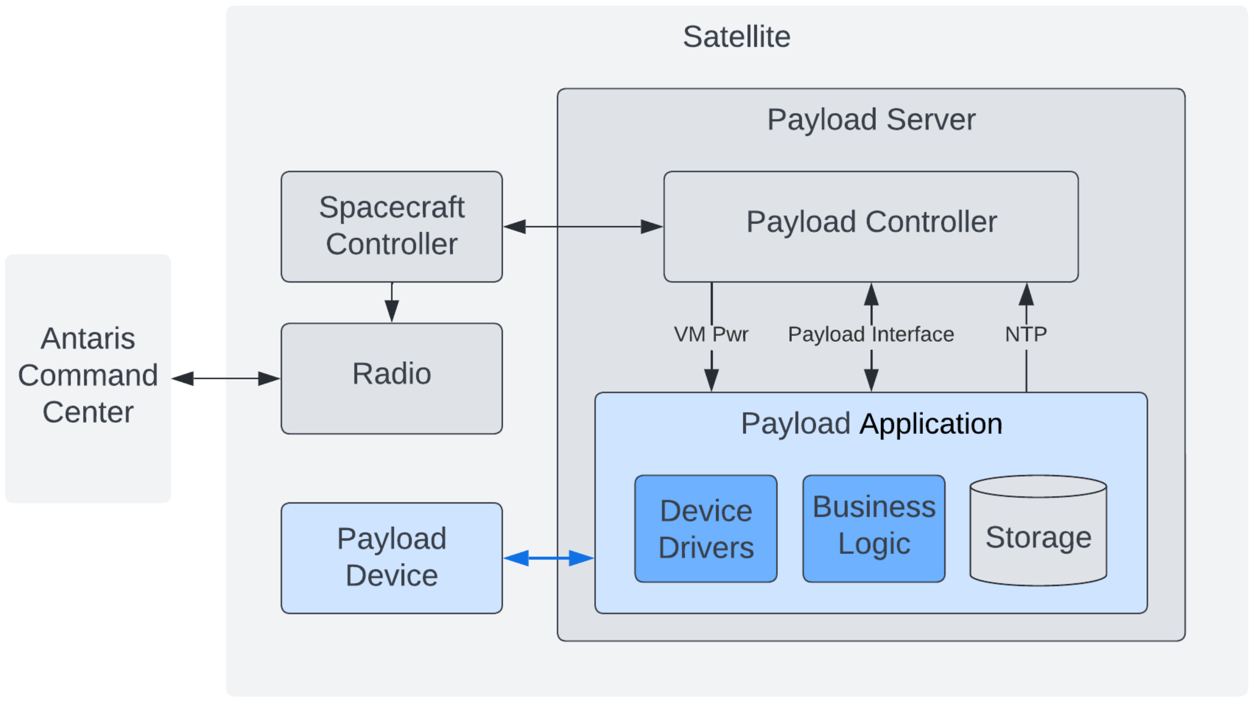
Payload Application (PA): Software developed by a Payload Developer using the Payload SDK which operates its Payload Device, interacts with PC for flight services and executes payload mission business logic. It is developed by payload owners to execute a known set of Payload Sequences.

Payload Sequence: a discrete operation designed by Payload Developers to support mission objectives. Payload Sequences are triggered by Telecommands.

Telecommand (TC): represents a command sent from an operator to a satellite, representing a logical action to be taken, for example “point towards the ground”, “establish X-band radio connection”, or “execute XYZ payload sequence”. Telecommands may be executed on-demand during a ground contact, or they may be scheduled for execution over time.

3.2. System Architecture

A payload developer is responsible for a Payload Application (PA) and a Payload Device – effectively everything highlighted in blue below:



Payload Applications contain payload-specific business logic and any device drivers necessary to interact with a Payload Device. Storage is made available to the application that persists across reboots or unexpected failures.

While active, a Payload Application has full control over its associated device and may interact with the Spacecraft Controller via the Payload Controller and the "Payload Interface". This allows Payload Applications to coordinate with the onboard scheduler, emit telemetry/teledata for downlink, implement health checking, etc.

The Spacecraft Controller manages all communication with the Antaris Command Center (i.e. mission control services), typically via S- and X-band radio links. This is the control and data path for onboard payloads.

So-called "massless" Payloads are also supported by this architecture. In this case, a Payload Application is simply developed without an associated Payload Device.

4. Application Design

A "Payload Application" encompasses all software operating within the Payload Server for a specific payload. This includes application-specific software, system libraries, and even the encapsulating VM.

This section of the SDK guide explains how a developer should think about design and implementation of a Payload Application.

4.1. Application Operation

When a Payload Application is scheduled to be active for a given period of time, a simple state machine is implemented:

1) Payload Device is powered on
2) Payload Application is booted up
3) Payload Application starts (via entrypoint script)
4) Payload Application initializes a Payload Interface connection
5) Payload Controller instructs Payload Application to execute necessary Payload Sequences
6) After sequences are executed, the PA shuts down
7) Payload Device is powered off

4.2. Application Modes

On boot, the Payload Application has an opportunity to determine the “mode” of operation requested. This is used to instruct the Payload Application to start up in one or more states to facilitate actions such as upgrading application software or implementing a “factory reset” to recover from some failure. This “mode” check would happen within step 2) and alter steps 3) through 6) as described above.

4.3. Payload Sequences

Payload developers model executable routines as “Payload Sequences”. A sequence represents a discrete unit of work, and usually maps to specific manipulation of a payload device in accordance with mission objectives.

Payload Sequences are scheduled using simple heuristics, all controlled by the Antaris team. Examples of this include “when the satellite is within range of X,Y coordinates” or “twice per orbit during the local day time”.

There is no support for dynamic pointing at this time (i.e. manipulation of ADCS to point at targets determined by payload application at time of execution), but this will be supported in a future SDK release. Geographic triggers are considered by the Antaris team when generating spacecraft schedules.

A Payload Application is instructed to execute a sequence using the Payload Interface (documented below). Payload Applications are not “always on”, and will only be booted up when its sequences are to be executed. Sequences are always given a duration within which they are expected to run, and are not able to run “forever”.

Dynamic/on-demand interaction for active debugging and diagnosis is supported directly by the Antaris team. This is not discussed further in this document.

4.4. Tasking & Scheduling

A **Task** represents a higher-level Payload CONOPS operation, such as spacecraft station-keeping or payload manipulation. Tasks take the form of templates containing ordered sets of Telecommands. For example, a Task might be defined to point to a location on the ground and manipulate an earth observation payload:

Example Task "exec_payload_imager" (Duration = D)			
#	**Telecommand**	**Parameters**	**Time**
1	`adcs_point_nadir`		T
2	`power_on_payload`	`Payload_HW_ID=7`	T1
3	`boot_payload_app`	`Payload_APP_ID=4, Mode=primary`	T2
4	`start_sequence`	`Payload_APP_ID=4, Seq_ID="B",` `Seq_Params="arg1 arg3 etc", Duration=D`	T3
5	`shutdown_payload_app`	`Payload_APP_ID=4`	T3+D

An Operator uses Tasks to construct a **Schedule.** Schedules contain a series of Telecommands rendered from input Tasks. A schedule typically spans one or more days, beginning some number of hours or days in the future. This is used to instruct a satellite how to autonomously operate while outside of an active ground station contact.

An example Schedule could be created from the following tasks, taking the provided Start Time and Duration as input:

#	**Task**	**Start Time**	**Duration**
1	`execute_payload_imager`	02:00:00	1200
2	`ground_contact_alaska`	02:24:40	600

The rendered Schedule might look like so:

#	**Telecommand**	**Parameters**	**Time**
`execute_payload_imager(D=1200)`			
1	`adcs_point_nadir`		02:00:00
2	`power_on_payload`	`Payload_HW_ID=7`	02:02:30
3	`boot_payload_app`	`Payload_APP_ID=4, Mode=primary`	02:04:00
4	`start_sequence`	`Payload_APP_ID=4, Seq_ID="B",` `Seq_Param="arg1 arg3 etc", Duration=1200`	02:04:30
5	`shutdown_payload_app`	`Payload_APP_ID=4`	02:24:30
`ground_contact_alaska(D=600)`			
6	`adcs_point_lat_lng`	`lat_lng=61,-147`	02:24:40
7	`execute_ground_contact`	`bands=s,x`	02:26:00

Task definition and scheduling is a collaborative, ongoing exercise with the Antaris team. During initial payload development, it is helpful to keep the following dimensions in mind:

Name	Description
Task ID	An alphanumeric value assigned by payload developer.
Trigger Conditions	Location, time of day, etc
Task Duration	Amount of time this task is desired to run
Executions per Orbit/Day	Number of times this Task should be invoked within a given time period, likely per orbit or per 24-hour period
Power Requirements	Avg and max power required for the Task
Payload Device State	Expected payload device power state before and during Task execution

4.5. File Upload / Download

File uploads are handled directly by the Antaris team on behalf of a Payload Developer. Uploaded files are made available at a pre-determined location in a Payload Application's storage space. Keep in mind that radio uplink bandwidth is relatively limited, so it is wise to minimize upload file size and to consider piecemeal update processes during Payload Application development.

File downloads are typically initiated in response to creation of some mission-oriented data by the Payload Application and/or Payload Device. A PA must inform the satellite that files are ready to be downloaded using the Payload Interface (documented below). Files will be distributed by the Antaris team to developers once on the ground.

4.6. Application Upgrades

Payload Applications are expected to upgrade themselves, typically via debian package updates.

An alternate PA mode should be used to trigger an upgrade. This explicit approach is preferred as it allows for upgrade/recovery in the event the PA is unable to operate normally.

A typical upgrade flow would look like so:

1. Payload Developer coordinates with Antaris to upload needed file(s) to Payload Application storage
2. A Telecommand is issued to Payload Controller to boot the Payload Application into an "upgrade" mode
3. On boot, the PA entrypoint.sh script detects the alternate mode of operation and applies any software/filesystem changes necessary within the VM
4. A Telecommand is issued to Payload Controller to shut the Payload Application down
5. Later Telecommands would then boot the PA into a nominal mode and resume normal operations.

Above flow can be built as a scheduled Task.

Upgrade of application's entrypoint.sh script will be handled by Payload Controller.

5. Using the Payload SDK

Proper Payload Application development relies heavily on the Payload SDK, which contains necessary libraries, tools, documentation, and other assets.

Artifacts in the SDK include:

- README.md file, which details out how to use this SDK
- Payload Application Artifacts
    - A Virtual Machine base image or Container with required dependencies installed
    - API libraries and applicable header files (C++, Python)
    - Sample payload applications (C++, Python), showcasing usage of Payload Interface
    - Sample entrypoint.sh script
    - Sample environment config (config.json) file (see section 6)
    - Sample debian package spec file (to create application package)
- Payload-Controller Artifacts
    - Simulated Payload Controller Software (C++)
    - Sample Payload Tasks-Schedule json file (given as input to the Payload-Controller)
- Other tools to aid in the development and testing process

A developer MUST use an SDK library to automate many lower level details of PA operation. Current language support is limited to Python and C++, but support for other languages is planned.

5.1. Developer Deliverables

Developers are expected to deliver a debian software package (.deb file) which bundles their payload application software, an entrypoint script, and necessary dependencies. Antaris will install this package in the VM provisioned for the application.

6. Application System Environment

6.1. Configuration

Application configuration is provided via the readonly **`/opt/antaris/app/`** directory in VM :

- **`config.json`** contains a JSON-encoded config file, constructed by the PC to help automate PA configuration. This file is subject to change until final release before satellite launch.
- **`mode`** contains the active application "mode". This file is typically read by the entrypoing script to enable high-level switching in behavior of PA early in the startup process

6.2. Compute

All Payload Applications are deployed as Virtual Machines (VMs). CPU and memory resources are configured based on requirements shared with the Antaris team.

6.3. Storage

Storage capacity is pre-configured. All storage is persistent, and will maintain state across reboots. Access to storage is provided via the following filesystem mounts:

- **`/opt/antaris/outbound/`** produced by the PA that are intended for download to Antaris Command center during a ground station pass
- **`/opt/antaris/inbound/`** contains files uploaded from Antaris Command Center to the PA. Application will have read-only access of this folder
- **`/opt/antaris/app/`** contains –
    - environment config file (config.json) – with read-only access
    - application mode file (mode) – with read-only access
    - entrypoint script file (entrypoint.sh)
- **`/opt/antaris/sdk/`** contains antaris payload SDK bundle
- **`/opt/antaris/workspace/`** is intended to be used as a “workspace”, supporting active operation of a PA

6.4. Network

Each Payload Application receives a unique IP Address, as does the associated Payload Device. The Payload Controller and an NTP server are also available over this network. The values assigned to these resources are defined in the PA config, and may be accessed via the SDK library.

6.5. I/O Interface

The exact I/O interfaces used by a given payload are subject to joint discussion/design with Antaris. Only when required for specific Payload Device access:

- Required GPIO pins are mapped as files
- Required UART ports are mapped as devices

Exact locations of these I/O resources are defined in the app config.

7. Payload Interface

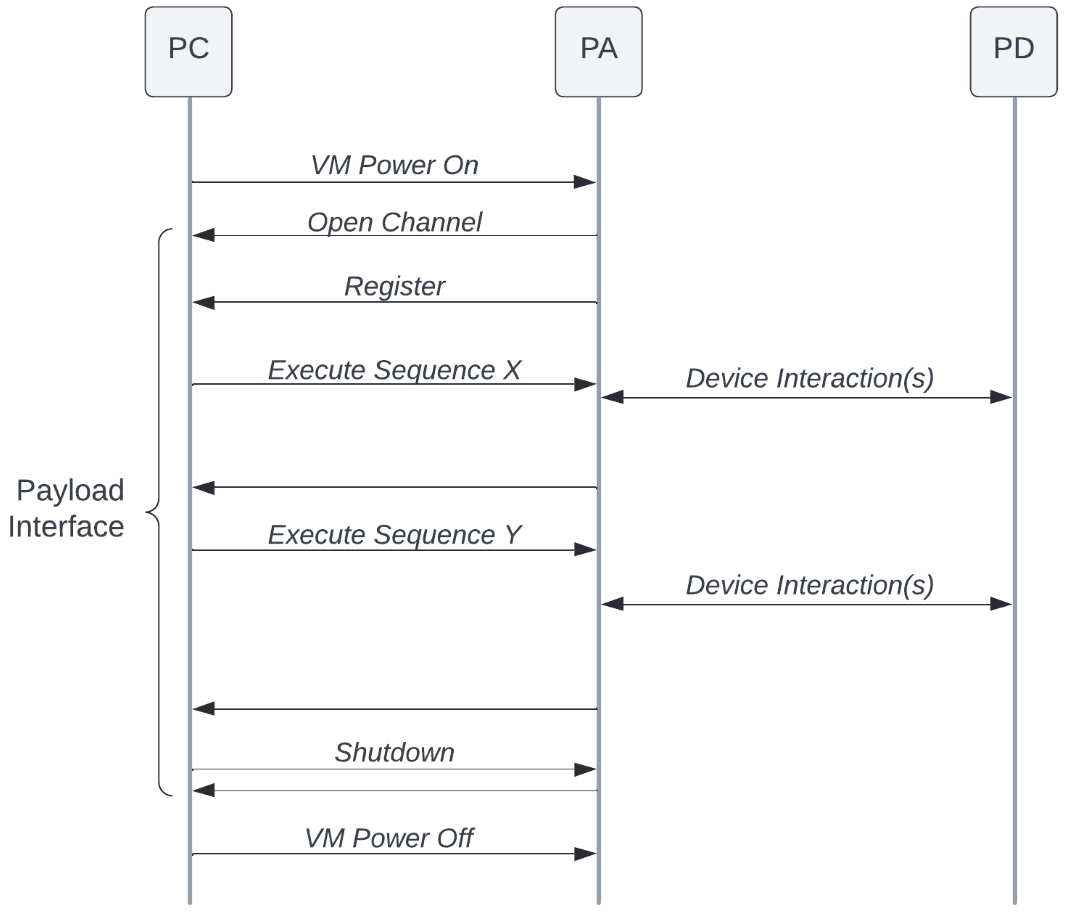
The Payload Interface is a bidirectional communication mechanism utilized by the Payload Controller and a Payload Application. A “channel” below represents an open connection between a single PA and the PC.

The Payload Interface relies on an asynchronous request/response model. A “correlation ID” is used to map responses back to the originating requests when async responses are to be expected. The Payload Controller also works with this asynchronous model, expecting async responses from the Payload Application to its own requests.

Though the API schema and data structures defined in this document are in syntax of programming language C, the SDK library is available in multiple languages.

7.1. Request Flow Example

The following flow diagram represents how the Payload Interface is used during a typical period of payload operation:



7.2. Encryption & Authentication

Communication between a Payload Application and the Payload Controller is encrypted. All messages between the PA and PC are encrypted. This encryption feature is built into available SDK software libraries, and should be left enabled at all times.

Authentication of communication via the Payload Interface is achieved with a simple auth token retrieved during the PA registration call. This detail is handled by the SDK software libraries made available to payload developers, and no explicit action needs to be taken.

7.3. Establishing a Connection

To begin communication via the Payload Interface, PA creates a connection to the Payload Controller via the *CreateChannel* API. With the *CreateInsecureChannel* API, communication between PA and PC will not be encrypted. Once a channel is created, it is used to facilitate all other API calls.

To explicitly close the channel, a PA may use the *DeleteChannel* API. This is not required, but highly encouraged as it helps the PC understand the current state of the PA.

7.4. Payload Application Requests

Below are the interactions that may be initiated by a Payload Application. Note that most requests define a corresponding asynchronous response, but some do not.

7.4.1. Register Payload Application

Register the PA with the PC, typically executed immediately after opening a channel. The PC will only communicate with PAs that successfully register themselves. After some time without registration, a PA will be considered in a bad state and may be shut down prematurely.

Request	Register
Parameters	U16 CorrelationId; U16 HealthCheckFailAction; // 0 => No action, 1 => Reboot

HealthCheckFailAction indicates what the PC should do if the application fails to respond to health checks five consecutive times.

7.4.1.1. Expected async response

Response	ResponseRegister

Parameters	U16 CorrelationId; // Same as what was sent in Register API
	U16 ReqStatus; // SUCCESS (0) or Error Code (non-zero)
	[256]U8 AuthToken;

7.4.2. Get Current Location

Retrieve current GNSS (GPS) location.

Request	GetCurrentLocation
Parameters	U16 CorrelationId;

7.4.2.1. Expected async response

Response	ResponseGetCurrentLocation

Parameters	U16 CorrelationId; // Same as what was sent in GetCurrentLocation API
	U16 ReqStatus; //SUCCESS (0) or Error Code (non-zero)
	FLOAT Latitude;
	FLOAT Longitude;
	FLOAT Altitude;
	U64 DeterminedAt; //Epoch time when the location was determined

The *DeterminedAt* parameter is important to consider, as there will be a delay from the time at which location data was collected and the time at which the PA receives that data.

7.4.3. Set Payload Device Power State

Request to change the power state of the Payload Device. PA can power cycle its device by issuing a Power-Off request followed by a Power-On after some delay.

Request	PayloadPowerControl
Parameters	U16 CorrelationId; U16 PowerOperation; // 0=Power-Off, 1=Power-On

7.4.3.1. Expected async response

Response	ResponsePayloadPowerControl

Parameters	U16 CorrelationId; // Same as what was sent in PayloadPowerControl API
	U16 ReqStatus; // SUCCESS (0) or Error Code (non-zero)

7.4.4. Stage File Download

Indicate that a file is ready to be download through a ground station. Payload Application needs to copy the file into `/opt/antaris/outbound` before making this API call. Payload Controller will delete this file from the outbound folder once its downloaded to ground station successfully.

Request	StageFileDownload

Parameters	U16 CorrelationId;
	char[64] FileLocation;

FileLocation should be a relative path within `/opt/antaris/outbound`. For example, to stage a file located at `/opt/antaris/outbound/foo/bar.json`, one would set `FileLocation` to `foo/bar.json`.

7.4.4.1. Expected async response

Response	ResponseStageFileDownload

Parameters	U16 CorrelationId; // Same as what was sent in StageFileDownload API
	U16 ReqStatus; // SUCCESS (0) or Error Code (non-zero)

ReqStatus=SUCCESS means the file has been staged for download successfully and its transfer to ground station will start at some later point in time.

7.5. Payload Controller Requests

Below are the interactions that may be initiated by the Payload Controller. Responses, when appropriate, are sent from the Payload Application.

7.5.1. Start Sequence

PC sends this command to the PA to instruct it to execute a known sequence immediately.

Request	StartSequence

Parameters	U16 CorrelationId;
	char[16] SequenceName; // Alphanumeric string, understood by PA
	char[64] SequenceParams; // Alphanumeric string, understood by PA
	U64 ScheduledDeadline; // Deadline (in epoch) for sequence handling

ScheduledDeadline is the absolute UNIX time by which the sequence must be completed.

7.5.1.1. Expected async response

Request	SequenceDone
Parameters	U16 CorrelationId; // Same as what came in SequenceStart request

7.5.2. Shutdown

Initiate a PA shutdown immediately. Application can shutdown its payload hardware gracefully before shutting itself down. PA will have a graceful shutdown deadline, the length of which is pre-configured while preparing CONOPS Task schedule.

Request	Shutdown

Parameters	U16 CorrelationId;
	U64 ShutdownDeadline;

ShutdownDeadline is the absolute UNIX time at which the PA must must have issued a response and shut down gracefully, otherwise more aggressive shutdown procedures may be taken.

7.5.2.1. Expected async response

Response	ResponseShutdown

Parameters	U16 CorrelationId; // Same as what PC sent in Shutdown request
	U16 ReqStatus; // SUCCESS (0) or Error Code (non-zero)

7.5.3. Health Check

PC monitors PA health by calling this request periodically (every 5 seconds). The PA should be prepared to process this request and report an accurate representation of its health.

Request	HealthCheck
Parameters	U16 CorrelationId;

7.5.3.1. Expected async response

Response	ResponseHealthCheck

Parameters	U16 CorrelationId; // Same as what PC sent in Health Check request
	U16 AppState; // Zero means Good, Non-Zero means in-Error
	U16 ReqsToPCInErrCnt; // Count of requests to PC that faced error
	U16 RespsToPCInErrCnt; // Count of responses to PC that faced error