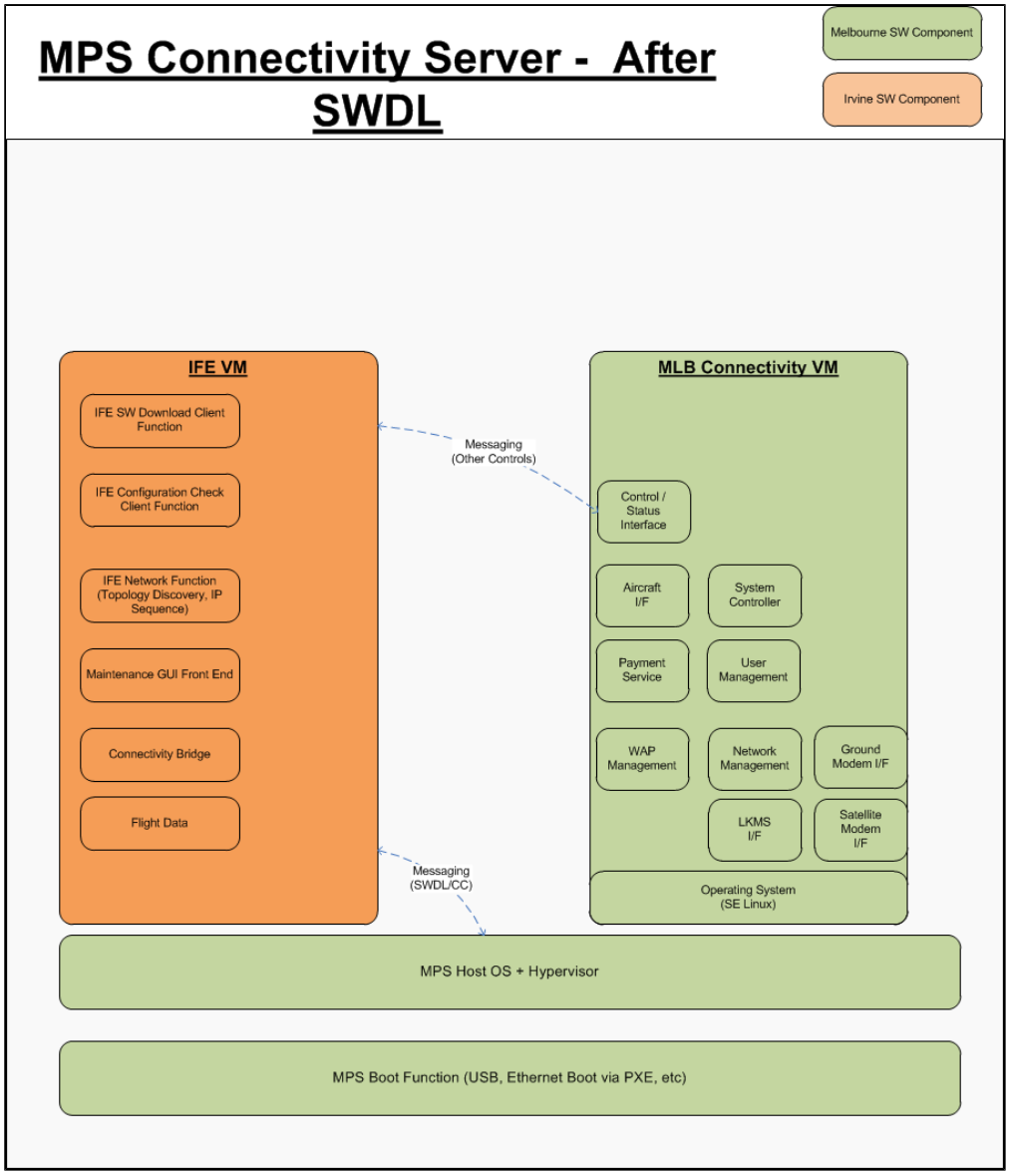
MPS Connectivity Platform ICD for Thales VAR Services

Overview

The Thales MultiPurpose Server (MPS), developed by Thales LiveTV (Melbourne, FL), is an Intel i7-based LRU capable of being used in conjunction with the Avant IFE system as a low-cost replacement for the DSU-D4C. When deployed in such a configuration, it will contain software components developed by Thales Melbourne, as well as by Thales Irvine, using a virtual machine (VM) model to encapsulate functionality provided by each team.

- The Connectivity VM, developed and maintained by Thales Melbourne, will contain interfaces for control and status for the connectivity hardware, including satcom terminals, and wireless access points (WAP), as well as connectivity-related software components enabling service to passengers via the Thales Value-Added Reseller (VAR) on the Inmarsat Global Xpress (GX) network. Software running in this environment will include components responsible for satcom link initialization and monitoring, network traffic control, WAP radio control and management, and VAR services.
- The Irvine GX VM (previously known as the IFE VM) and its software components will be developed and maintained by Thales Irvine, and will contain functionality necessary to interface with the Avant IFE system and maintenance aspects specific to the GX satcom system. Such functionality will include the crew and maintenance GUIs (front-end and back-end logic), the 615A software download mechanism of the GX system, collection of event logs from the GX system, propagation of IFE flight data, and the reporting of BITE faults to the IFE system.
- Additionally, the MPS Application Platform (MAP) services on the host OS will provide a set of secure interfaces such as access to hardware or Low-level Software (LLS) abstractions, available to software components running inside VMs. For example, software download, config check, and keyline control operations are some of the interfaces provided by MAP.



Scope

This document, developed with input from both Thales Irvine and Thales Melbourne, exists to define and capture the specific set of software interfaces which must exist between components in order to facilitate integration of the MPS as a connectivity server (MPS-C) into an Avant IFE environment. Its primary focus is on defining the overall messaging patterns and specific message details for the interface between the Connectivity VM and the Irvine GX VM, known as the Trusted Host Interface (THI).

Technology

While defining these interfaces, one of the goals was to be able to maximize reuse of existing software components and knowledge, while maintaining a loose coupling between the software written by the Irvine and the Melbourne connectivity teams. Since both teams already have experience developing software using Google Protocol Buffers and ZeroMQ and technologies, these will serve as the foundation for data interchange between software components maintained by the two teams.

Google Protocol Buffers v2

Protocol Buffers are a language and platform neutral method of serializing structured data into an efficient format for use in network communications. Because its design goals emphasize simplicity and performance, it is a good alternative to structured text-based formats (such as XML or JSON). Data structures (called messages) are defined in a proto definition file using the proto2 interface description language (IDL) syntax. These are then compiled to generate necessary code in any supported language, such as methods to serialize and parse messages, and get or set message field values. In operation, messages are serialized into a binary wire format which is forwards and backwards compatible due to the use of unique numbered tags to identify message fields. These serialized messages are not human readable, nor are they self-describing, so the receiver must have knowledge of the message type in order to properly decode the data.

ZeroMQ 4

ZeroMQ (ZMQ) is a library providing high-performance asynchronous messaging between software components, without requiring a dedicated message broker process. The API provides socket abstractions (connected using one of several lower-level transports) capable of implementing one-to-many connections between endpoints, in order to send or receive multi-part messages atomically. The network topology differs depending on the pattern implemented, but the basic patterns of request-reply, publish-subscribe, push-pull, and exclusive pair can be combined in many ways, resulting in a simple yet flexible architectural model.

Security

With the release of ZeroMQ 4.0, support was added for a new security framework, based on the IETF's Simple Authentication and Security Layer (SASL), which allows the negotiation of a security mechanism between client and server before exchanging any other data. The security mechanism are extensible, however three are provided by default: **NULL** (classic ZeroMQ, with no authentication or encryption), **PLAIN** (plain-text username and password authentication, no encryption), **CURVE** (secure authentication and encryption based on elliptic curve cryptography, using the Curve25519 algorithm). Depending on the specific security needs of the interface being implemented, we can use these mechanisms to implement one of the following standard ZMQ security patterns (see http://hintjens.com/blog:49 for details and example code):

ZMQ Pattern Name	ZMQ Security Mechanism	Source IP Filtering	Client Authentication	Encryption	Description
Grasslands	NULL	No	No	No	Plain text with no security at all (like legacy ZeroMQ sockets)
Strawhouse	NULL	Yes	No	No	Plain text with filtering based on connection source IP addresses

Woodhouse	PLAIN	Yes	Plain-text username/password	No	Plain text, with IP filtering, and a simple username and password check
Stonehouse	CURVE	Yes	Server's Public Key	Yes	Encrypted, with IP filtering, and server authentication (clients must know the server's public key)
Ironhouse	CURVE	Yes	Server's Public Key + Client's Public Key	Yes	Encrypted, with IP filtering, and mutual server and client authentication (server and client must both know the other's public key)

Trusted Host Interface

Common Data Types

This section defines the common Google Protocol Buffer messages and enums that are shared across the different interfaces.

Property

This structure is used for defining a key/value pair and has the following structure:

Name	Status	Type	Description	Example
key	REQUIRED	string	The name (or the key) of the value.	satcom.satlink
value	REQUIRED	string	The value associated with this key.	enabled

IDL:

```
message Property
{
```

```
   required string key = 1;
   required string value = 2;
}
```

Error Message

For request/response type messages if there was an error processing a request the following error message will be included in the response:

Name	Status	Type	Description	Example
error_code	REQUIRED	uint32	A unique error code for this error on the particular interface.	23
error_description	OPTIONAL	string	The value associated with this key.	"WAP Comm Failure"

IDL:

```
message ErrorMsg
{
   required uint32 error_code = 1;
   optional string error_description = 2;
}
```

Common Error Codes

The following tables lists some common error codes and descriptions that are used across multiple interfaces.

Error Name	Error Code	Error Description
INTERNAL_ERROR	1	There was an unexpected internal error processing the request.
SERVICE_UNAVAILABLE	2	The requested service is currently unavailable.
UNSUPPORTED_OPERATION	3	The requested operation is unsupported on the current configuration of hardware and software.
RESETTING	4	The requested operation could not be completed because a portion of the system is currently resetting.
INVALID_PARAMETERS	5	The specified request contained invalid parameters.

Network Design

The subnet **192.168.10.0/24** is paired with **VLAN310** and the following configuration is required:

- **Irvine GX VM**: VLAN310 interface has IP address of **192.168.10.1**, netmask 255.255.255.0
- **Portal VM**: VLAN310 interface has IP address of **192.168.10.2**, netmask 255.255.255.0
- **Connectivity VM**: VLAN310 interface has IP address of **192.168.10.254**, netmask 255.255.255.0

Message Format

The message format to be used when transmitting the ZeroMQ messages is multi-part, with dedicated frames for message name, a common message header, and message data:

1. The first part contains the Google Protocol Buffers message name, as a string. This message name allows the receiving client to know which message type is being sent in a given message, so that it can properly decode the subsequent Google Protocol Buffers binary data in the payload. Defining this as the first frame also has an additional advantage, because only this frame can be used for the ZeroMQ subscription topic filter. When used with PUB/SUB socket patterns, this allows a client to choose to receive only a subset of the available message types being broadcast by subscribing to one or more message names after the initial connection has been made.
    a. The second part contains a common message header for payload metadata, specified as key-value pairs, in Google Protocol Buffers binary format. As new header keys may be added in the future, clients should ignore any unknown header keys during processing.

    At the time of this writing, the following headers have been defined:

Name	Status	Type	Description	Example	Notes
msg_name	REQUIRED	string	The name of the Google Protocol Buffers message in the payload frame.	BITEFault	This the same string as what's in the first frame of the ZMQ message.
source	OPTIONAL	string	An optional string indicating the name of the source of the message.	IFEVM	
destination	OPTIONAL	string	An optional string indicating the destination of the message.	Connectivity VM	

sequence_nbr	OPTIONAL	uint64	An optional sequence number indicating the numbe of unique instances of this message have been sent.	1003	If the same message is re-transmitted for some reason then the same sequence number should be used.
retransmission	OPTIONAL	boolean	"true" if this message is a re-transmission of the previous message with no changes. Otherwise "false".	true	
add_data	REPEATED	Property	An optional list of Property key/value pairs.	[{"key1":"value1"},{"key2":"value2"}]	

IDL:

```
message MsgHeader
{
  required string msg_name = 1;
  optional string source = 2;
  optional string destination = 3;
  optional uint64 sequence_nbr = 4;
  optional boolean retransmission = 5;
  repeated Prototype add_data = 6;
}
```

2. The third part contains the message payload itself, in Google Protocol Buffers binary format.

Irvine GX VM

The Irvine GX VM of the MPS offers interfaces providing near real-time aircraft flight data and IFE system mode information to third party software components, as well as interfaces permitting software components to transmit faults and events to be recorded by the Built-In Test Equipment (BITE) software on the Avant IFE system head-end. Software components running inside the Irvine GX VM act as a bridge or gateway to the Avant IFE system, and are responsible for relaying data between the Avant system and any software components on the MPS which need to integrate with it.

Aircraft Flight Data and IFE System Mode

Periodic updates of aircraft flight data and IFE system mode status are provided, using data is obtained from the Avant IFE system via an AMQP connection to the RabbitMQ message broker. The data is shared with

other software components running on the MPS, such as in the Connectivity VM, using a ZeroMQ PUB socket associated with a well-known IP and port on the Irvine GX VM.

Usage Summary

<table>
<tr><td>Design Pattern</td><td>ZMQ XPUB/SUB, with Last Value Caching (LVC)</td></tr>
<tr><td>Security Pattern</td><td>Ironhouse</td></tr>
<tr><td>Server Socket</td><td>Irvine GX VM bind() of XPUB socket on well-known 192.168.10.1:28505 (with ZMQ_XPUB_VERBOSE sockopt)</td></tr>
<tr><td>Client Socket(s)</td><td>connect() of SUB socket using an ephemeral port (from client such as Connectivity VM)</td></tr>
<tr><td>Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data
<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td><code>AircraftData</code></td></tr>
<tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>2*OCTET (see ProtoBuf Encoding)</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>2*OCTET (see ProtoBuf Encoding)</td></tr>
</table>
</td></tr>
</table>

Pattern Details

The connectivity bridge component on the Irvine GX VM acts as a forwarding proxy, relaying received Google Protocol Buffers message payloads from AMQP to a local ZeroMQ XPUB socket bound to a well-known IP address and port number. Software components wishing to receive aircraft flight data and IFE system mode information connect using a corresponding ZeroMQ SUB socket, and subscribe to one or more message types (shown in detail below). Because the data is not sensitive, no connection authentication is required, and the message data will be transmitted in plaintext - this is known as the Grasslands ZeroMQ security pattern (and works exactly like legacy ZeroMQ sockets).

The data flow using the (X)PUB/SUB ZeroMQ pattern is primarily one-way, from the server socket to the client socket, with the exception of the client subscription messages. These subscription messages can be received directly when using the XPUB socket variant, allowing the server to be aware of client subscription actions. The design of the ZeroMQ (X)PUB socket allows multiple clients to be connected simultaneously, in which case any messages written to the socket will be sent to all connected clients which have subscribed

to that message type. In the case where there are no connected clients, messages written to the socket will be silently discarded.

Last Value Caching

Last Value Caching (LVC) solves the problem of how a new subscriber, upon connection to a ZMQ publisher socket, catches up with the state of messages which have previously been transmitted. Using the ZeroMQ socket variant of XPUB, the ZMQ publisher is notified when a new subscriber joins and subscribes to some specific message types. It can then rebroadcast the last message received for each of these types to all connected subscribers on the publisher socket. By including a header providing the sequence number of the last update received for the specific message type (see "Message Format" above), the receiving client is then able to determine whether the message is simply a re-transmission, and should be ignored.

The following steps provide an overview of using the last value caching pattern to publish aircraft flight data:

1. The connectivity bridge SW running on Irvine GX VM:
    a. Connects to the IFE AMQP exchange to receive flight data.
    b. Establishes a ZMQ XPUB socket listener (using the ZMQ_XPUB_VERBOSE sockopt allows potentially duplicate ZMQ subscription events to be received as well).
    c. Receives flight data messages via AMQP and locally stores the last message payload for each message type (eg: "AircraftData"), as well as the current sequence number from the common message header.
    d. Publishes the received message data to all connected clients via the XPUB socket.
2. An MPS client SW component, intending to receive flight data events:
    a. Creates a new ZMQ SUB socket.
    b. Connects to the connectivity bridge SW XPUB socket.
    c. Sets one or more ZMQ_SUBSCRIBE sockopt message filters on the socket, with the message types that it wants to receive (eg: "AircraftData"). Multiple subscribe filters may be defined for a given SUB socket. If an empty subscribe filter is provided, this has the effect of causing all message types to be received.
3. On the connectivity bridge SW:
    a. A separate ZMQ subscription message is generated by the XPUB socket for each message subscribe request received from a client.
    b. The stored values corresponding to the subscription message(s) are immediately published on the XPUB socket, along with the sequence number of the last update in the common message header.
4. The MPS client SW component (and all other connected clients):
    a. Receives the latest available message for each subscribed message type via the connected SUB socket.
    b. Continues to receive ongoing message updates for each subscribed message type via the connected SUB socket.

Message Definitions

Aircraft Data

The *AircraftData* message provides static information related to the aircraft which does not change during a flight.

ZMQ Frame No.	Data
0	`AircraftData`
1	MsgHeader

2	Google Protocol Buffers message data in binary format, using the following message definition:
	
	`message AircraftData {`
	`enum AircraftType {`
	`UNKNOWN = 0;`
	`A310 = 1;`
	`A318 = 2;`
	`A319 = 3;`
	`A320 = 4;`
	`A321 = 5;`
	`A330 = 6;`
	`A340 = 7;`
	`A350 = 8;`
	`A380 = 9;`
	`B737 = 10;`
	`B747 = 11;`
	`B757 = 12;`
	`B767 = 13;`
	`B777 = 14;`
	`B787 = 15;`
	`C200 = 16;`
	`C700 = 17;`
	`C705 = 18;`
	`C900 = 19;`
	`E170 = 20;`
	`E175 = 21;`
	`E190 = 22;`
	`E195 = 23;`
	`}`
	`enum AircraftManufacturer {`
	`UNKNOWN = 0;`
	`AIRBUS = 1;`
	`BOEING = 2;`
	`BOMBARDIER = 3;`
	`EMBRAER = 4;`
	`COMAC = 5;`
	`}`
	`enum Validity`
	`{`
	`UNKNOWN = 1;`
	`UP_TO_DATE = 2;`
	`OUT_OF_DATE = 3;`
	`UP_TO_DATE_OVERRIDEN = 4;`
	`COMPUTED = 5;`
	`}`

	`// Aircraft Type`
	`required AircraftType type = 1 [default = UNKNOWN];`
	`optional Validity type_validity = 6 [default = UNKNOWN];`
	
	`// Aircraft Manufacturer`
	`required AircraftManufacturer manufacturer = 2 [default = UNKNOWN];`
	`optional Validity manufacturer_validity = 7 [default = UNKNOWN];`
	
	`// Tail Number - up to 12 alphanumeric characters.`
	`optional string tail_number = 3;`
	
	`// Nose Number (Fleet Number) - up to 12 alphanumeric characters.`
	`optional string nose_number = 4;`
	
	`optional string ICAO = 5;`
	`}`

Flight Leg Data

The *FlightLegData* message provides departure and arrival information.

ZMQ Frame No.	Data
0	`FlightLegData`
1	MsgHeader

2	Google Protocol Buffers message data in binary format, using the following message definition:
	
	`message FlightLegData {`
	`enum Type {`
	`REAL = 0;`
	`TEST = 1;`
	`ENGINEERING = 2;`
	`}`
	
	`enum Validity`
	`{`
	`UNKNOWN = 1;`
	`UP_TO_DATE = 2;`
	`OUT_OF_DATE = 3;`
	`UP_TO_DATE_OVERRIDEN = 4;`
	`COMPUTED = 5;`
	`}`

```
// Describes whether this is a
real flight, a test flight, or an
engineering flight.
required Type type = 1 [default =
REAL];

// Flight Number
required string flight_number = 2;
optional Validity
flight_number_validity = 23
[default = UNKNOWN];

// FMC Arrival/Departure Airport
Codes
required string
fmc_departure_airport = 3;
optional Validity
fmc_departure_airport_validity =
24 [default = UNKNOWN];
required string
fmc_arrival_airport = 4;
optional Validity
fmc_arrival_airport_validity = 25
[default = UNKNOWN];

// Crew-Provided
Arrival/Departure/Reroute Airport
Codes
optional string
crew_departure_airport = 5;
optional Validity
crew_departure_airport_validity =
26 [default = UNKNOWN];
optional string
crew_arrival_airport = 6;
optional Validity
crew_arrival_airport_validity = 27
[default = UNKNOWN];
optional string
crew_reroute_arrival_airport = 7;
optional Validity
crew_reroute_arrival_airport_valid
ity = 28 [default = UNKNOWN];
```

```
// FMC Departure/Arrival City
Information
optional string fmc_departure_city
= 8;
optional Validity
fmc_departure_city_validity = 29
[default = UNKNOWN];
optional string
fmc_departure_longitude = 9;
optional Validity
fmc_departure_longitude_validity =
30 [default = UNKNOWN];
optional string
fmc_departure_latitude = 10;
optional Validity
fmc_departure_latitude_validity =
31 [default = UNKNOWN];
optional string fmc_arrival_city =
11;
optional Validity
fmc_arrival_city_validity = 32
[default = UNKNOWN];
optional string
fmc_arrival_longitude = 12;
optional Validity
fmc_arrival_longitude_validity =
33 [default = UNKNOWN];
optional string
fmc_arrival_latitude = 13;
optional Validity
fmc_arrival_latitude_validity = 34
[default = UNKNOWN];
```

```
// Crew-Provided
Departure/Arrival/Reroute City
Information
optional string
crew_departure_city = 14;
optional Validity
crew_departure_city_validity = 35
[default = UNKNOWN];
optional string
crew_departure_longitude = 15;
optional Validity
crew_departure_longitude_validity
= 36 [default = UNKNOWN];
optional string
crew_departure_latitude = 16;
optional Validity
crew_departure_latitude_validity =
37 [default = UNKNOWN];
optional string crew_arrival_city
= 17;
optional Validity
crew_arrival_city_validity = 38
[default = UNKNOWN];
optional string
crew_arrival_longitude = 18;
optional Validity
crew_arrival_longitude_validity =
39 [default = UNKNOWN];
optional string
crew_arrival_latitude = 19;
optional Validity
crew_arrival_latitude_validity =
40 [default = UNKNOWN];
optional string
crew_reroute_arrival_city = 20;
optional Validity
crew_reroute_arrival_city_validity
= 41 [default = UNKNOWN];
optional string
crew_reroute_arrival_longitude =
21;
optional Validity
crew_reroute_arrival_longitude_val
idity = 42 [default = UNKNOWN];
optional string
crew_reroute_arrival_latitude =
22;
optional Validity
crew_reroute_arrival_latitude_vali
dity = 43 [default = UNKNOWN];
}
```

Flight Phase Data

The *FlightPhaseData* message provides information about the various phases of the flight and the times they have occurred, and is updated approximately 9x per flight leg, or when any of the data items change.

ZMQ Frame No.	Data
0	`FlightPhaseData`
1	MsgHeader

2	Google Protocol Buffers message data in binary format, using the following message definition:
	
	`message FlightPhaseData {`
	`enum Phase {`
	`UNKNOWN = 0;`
	`PRE_FLIGHT_GROUND = 1;`
	`TAXI_OUT = 2;`
	`TAKEOFF = 3;`
	`CLIMB = 4;`
	`CRUISE = 5;`
	`DESCENT_APPROACH = 6;`
	`TOUCHDOWN = 7;`
	`TAXI_IN = 8;`
	`POST_FLIGHT_GROUND = 9;`
	`}`
	
	`enum Status {`
	`OPEN = 0;`
	`CLOSED = 1;`
	`}`
	`enum Validity`
	`{`
	`UNKNOWN = 1;`
	`UP_TO_DATE = 2;`
	`OUT_OF_DATE = 3;`
	`UP_TO_DATE_OVERRIDEN = 4;`
	`COMPUTED = 5;`
	`}`
	`// Flight Phase`
	`required Phase phase = 1 [default = PRE_FLIGHT_GROUND];`
	`optional Validity phase_validity = 16 [default = UNKNOWN];`
	`// Weight On Wheels (WOW)`
	`optional bool weight_on_wheels = 2 [default = true];`
	`optional Validity weight_on_wheels_validity = 17 [default = UNKNOWN];`

	`// Doors Open`
	`optional bool doors_open = 3 [default = true];`
	`optional Validity doors_open_validity = 18 [default = UNKNOWN];`
	
	`// IFE Flight Status`
	`required Status status = 4 [default = CLOSED];`
	
	`// Time at which the flight was opened/closed`
	`optional int64 open_time = 5;`
	`optional int64 closed_time = 6;`
	
	`// Time at which various flight phases occurred`
	`optional int64 pre_flight_ground_time = 7;`
	`optional int64 taxi_out_time = 8;`
	`optional int64 takeoff_time = 9;`
	`optional int64 climb_time = 10;`
	`optional int64 cruise_time = 11;`
	`optional int64 descent_approach_time = 12;`
	`optional int64 touchdown_time = 13;`
	`optional int64 taxi_in_time = 14;`
	`optional int64 post_flight_ground_time = 15;`
	`}`

Flight Navigation Data

The *FlightNavigationData* message provides information about the about the current aircraft and environmental conditions, and is updated every 1-2 seconds.

ZMQ Frame No.	Data
0	`FlightNavigationData`
1	MsgHeader
2	Google Protocol Buffers message data in binary format, using the following message definition:

```
message FlightNavigationData {
enum Validity
{
UNKNOWN = 1;
UP_TO_DATE = 2;
OUT_OF_DATE = 3;
UP_TO_DATE_OVERRIDEN = 4;
COMPUTED = 5;
}

// Time when this data was
generated or made available to IFE
Flight Data Service
optional int64 timestamp = 16;

// Current barometric altitude in
meters
optional string altitude = 1;
optional Validity
altitude_validity = 17 [default =
UNKNOWN];

// Current ground speed in km/h
optional string ground_speed = 2;
optional Validity
ground_speed_validity = 18
[default = UNKNOWN];

// Current mach speed in decimal
units - to 3 places - if available
optional string mach_speed = 3;
optional Validity
mach_speed_validity = 19 [default
= UNKNOWN];

// Current true air speed in km/h
- if available
optional string air_speed = 4;
optional Validity
air_speed_validity = 20 [default =
UNKNOWN];

// Current static air temperature
in degrees Celsius
optional string
static_air_temperature = 5;
optional Validity
static_air_temperature_validity =
21 [default = UNKNOWN];
```

```
// Current present position
latitude in decimal degrees (2
places)
optional string latitude = 6;
optional Validity
latitude_validity = 22 [default =
UNKNOWN];

// Current present position
longitude in decimal degrees (2
places)
optional string longitude = 7;
optional Validity
longitude_validity = 23 [default =
UNKNOWN];

// Current true heading in decimal
degrees (2 places)
optional string heading = 8;
optional Validity heading_validity
= 24 [default = UNKNOWN];

// Current true track angle in
decimal degrees (2 places)
optional string track_angle = 9;
optional Validity
track_angle_validity = 25 [default
= UNKNOWN];

// Current direction to
destination in decimal degrees (2
places)
optional string
direction_to_destination = 10;
optional Validity
direction_to_destination_validity
= 26 [default = UNKNOWN];

// Current wind speed in km/h
optional string wind_speed = 11;
optional Validity
wind_speed_validity = 27 [default
= UNKNOWN];

// Current true wind direction in
decimal degrees (2 places)
optional string wind_direction =
12;
optional Validity
wind_direction_validity = 28
[default = UNKNOWN];
```

	`// Remaining time from present position to destination, in 24 hour HH:MM:SS format`
	`optional string time_to_destination = 13;`
	`optional Validity time_to_destination_validity = 29 [default = UNKNOWN];`
	
	`// Remaining distance to destination, in meters`
	`optional string distance_to_destination = 14;`
	`optional Validity distance_to_destination_validity = 30 [default = UNKNOWN];`
	
	`// Estimated time of arrival, in 24 hour HH:MM:SS format`
	`optional string time_of_arrival = 15;`
	`optional Validity time_of_arrival_validity = 31 [default = UNKNOWN];`
	`}`

Date and Time Data

The *DateAndTimeData* message provides information regarding the current and elapsed time since departure, and is updated every 1 second.

ZMQ Frame No.	Data
0	`DateAndTimeData`
1	MsgHeader
2	Google Protocol Buffers message data in binary format, using the following message definition:

```
message DateAndTimeData {
enum Validity
{
UNKNOWN = 1;
UP_TO_DATE = 2;
OUT_OF_DATE = 3;
UP_TO_DATE_OVERRIDEN = 4;
COMPUTED = 5;
}

// Current UTC date and time in
YYYY-MM-DD HH:MM:SS format
optional string utc_date_and_time
= 1;
optional Validity
utc_date_and_time_validity = 8
[default = UNKNOWN];

// Elapsed time since departure in
HH:MM:SS format (based on
takeoff_time from FlightPhaseData)
optional string
time_since_departure = 2;
optional Validity
time_since_departure_validity = 9
[default = UNKNOWN];

// Local time at FMC departure
airport in YYYY-MM-DD HH:MM:SS
format
optional string
fmc_departure_date_and_time = 3;
optional Validity
fmc_departure_date_and_time_validi
ty = 10 [default = UNKNOWN];

// Local time at FMC arrival
airport in YYYY-MM-DD HH:MM:SS
format
optional string
fmc_arrival_date_and_time = 4;
optional Validity
fmc_arrival_date_and_time_validity
= 11 [default = UNKNOWN];

// Local time at crew-provided
departure airport in YYYY-MM-DD
HH:MM:SS format
optional string
crew_departure_date_and_time = 5;
optional Validity
crew_departure_date_and_time_valid
ity = 12 [default = UNKNOWN];
```

	// Local time at crew-provided arrival airport in YYYY-MM-DD HH:MM:SS format
	optional string crew_arrival_date_and_time = 6;
	optional Validity crew_arrival_date_and_time_validity = 13 [default = UNKNOWN];
	
	// Local time at crew-provided reroute airport in YYYY-MM-DD HH:MM:SS format
	optional string crew_reroute_arrival_date_and_time =7;
	optional Validity crew_reroute_arrival_date_and_time_validity = 14 [default = UNKNOWN];
	}

IFE System Mode

The *IFESystemMode* message provides the current IFE system mode, and is updated after a change.

ZMQ Frame No.	Data
0	IFESystemMode
1	MsgHeader

2	Google Protocol Buffers message data in binary format, using the following message definition:
	
	`message IFESystemMode`
	`{`
	`enum SystemModeType`
	`{`
	`UNKNOWN = 1;`
	`START_UP = 2;`
	`IDLE = 3;`
	`SERVICE = 4;`
	`DECOMP = 5;`
	`MAINT = 6;`
	`STATIC_TEST = 7;`
	`INTRUSIVE_TEST = 8;`
	`SW_DOWNLOAD = 9;`
	`CONTENT_DOWNLOAD = 10;`
	`IP_SEQ = 11;`
	`POWER_DOWN = 12;`
	`}`
	`// IFE System Mode`
	`required SystemModeType system_mode = 1 [default = UNKNOWN];`
	`}`

BITE Fault and Event Submission

The Avant IFE Built-In Test Equipment (BITE) suite of software detects and reports hardware and software faults for all components of the IFE system. On the MPS, software components running on the connectivity VM are responsible for continuously querying the MPS itself and related connectivity LRUs (such as the WAPs) for error conditions. As they occur, changes in fault status are transmitted using a ZeroMQ PUB socket associated with a well-known IP and port on the Connectivity VM. The BITE agent software component on the Irvine GX VM subscribes to these notifications using a ZeroMQ SUB socket, and submits them to the BITE software running on the Avant IFE head-end.

Usage Summary

Design Pattern	XPUB/SUB, with state summary caching
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of XPUB socket on well-known 192.168.10.254:49110 (with *ZMQ_XPUB_VERBOSE* sockopt)
Client Socket(s)	*connect()* of SUB socket using an ephemeral port (from client such as Irvine GX VM)

<table>
<tr><td>Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data
<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td><code>BITEFault</code></td></tr>
<tr><td>1</td><td>MsgHeader</td><td>2*OCTET (see ProtoBuf Encoding)</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>2*OCTET (see ProtoBuf Encoding)</td></tr>
</table></td></tr>
</table>

Pattern Details

A software component running on the Connectivity VM continually monitors and collects hardware and software BITE fault information from the MPS server and attached connectivity LRUs. When a change in fault status is detected, a Google Protocol Buffers message is generated indicating the LRU location, type or application name, fault identifier, and current fault state (along with optional data, such as a timestamp indicating when the fault was detected, or a short free-text description). These messages published to a local ZeroMQ XPUB socket bound to a well-known IP address and port number. Software components wishing to receive BITE fault data, such as the BITE agent on the Irvine GX VM, connect using a corresponding ZeroMQ SUB socket. Because the data is not sensitive, no connection authentication is required, and the message data will be transmitted in plaintext using the ZeroMQ Grasslands security pattern (see "ZMQ Security" above for more information).

Since the messages are sent using a (X)PUB/SUB pattern, the data flow is primarily from the server socket to the client socket, with the exception of the client subscription messages. Since the XPUB socket variant is used, these subscription messages can be received directly by the server, allowing it to be aware of client subscription actions. The design of the ZeroMQ (X)PUB socket allows multiple clients to be connected simultaneously, in case there are multiple software components needing access to BITE fault data, and any messages written to the socket will be transmitted to all connected and subscribed clients. In the case where there are no connected clients, messages written to the socket will be silently discarded.

When such a BITE fault message is received by the BITE agent software on the Irvine GX VM, the fault data is converted (if necessary) and transmitted to the IFE head-end using protobuf messages via an AMQP connection to the RabbitMQ message broker. The BITE agent running on the Irvine GX VM is responsible for maintaining an up-to-date configuration of possible BITE faults which may be reported by the MPS (to be provided by Melbourne), with a mapping each fault into a corresponding Avant IFE BITE equivalent. Having this conversion performed on the Irvine GX VM of the MPS allows the possibility for recording MPS-specific faults locally using the Thales fault code format.

State Summary Caching

The unofficial state summary caching pattern, similar in operation to LVC (see above), solves the problem

of how a new subscriber catches up with the state of BITE faults which have previously been transmitted. Using the ZeroMQ socket variant of XPUB, the ZMQ publisher is notified when a new subscriber connects and subscribes to any message type (the default state of the ZMQ SUB socket is to filter all incoming messages, so setting a subscription filter is effectily required).The publisher can then broadcast a special BITE fault summary message containing the current state of all known BITE faults, which is sent to every connected subscriber on the publisher socket. By using the sequence number in the common message header a receiving client is able to determine whether the message contains only repeated information, and should be ignored.

The following steps provide an overview of using the state summary caching pattern to publish BITE faults:

1. A software component running on the Connectivity VM:
    a. Monitors hardware and software faults from MPS and connectivity LRUs.
    b. Establishes a ZMQ XPUB socket listener (using the ZMQ_XPUB_VERBOSE sockopt allows potentially duplicate ZMQ subscription events to be received as well).
    c. Receives notification of fault events, generates BITE fault message payloads in Google Protocol Buffers format, and locally stores the last message payload for each unique BITE fault.
    NOTE: A unique BITE fault is a specific combination of LRU location, category/application name, fault index, and fault parameters.
    d. Publishes the generated BITE fault messages (*BITEFault*) to all connected clients via the XPUB socket.
2. A client SW component, intending to receive BITE fault events:
    a. Creates a new ZMQ SUB socket.
    b. Connects to the XPUB socket at the Connectivity VM.
    c. Sets a zero-length ZMQ_SUBSCRIBE sockopt message filter on the socket (eg: ""). An empty subscribe filter has the effect of causing all message types to be received.
3. On the Connectivity VM:
    a. A separate ZMQ subscription message is generated by the XPUB socket for each message subscribe request received from a client.
    b. A BITE fault summary message (*BITEFaultSummary*) containing the current state of all known faults, with the sequence number of the most recent fault in the MsgHeader, is immediately published on the XPUB socket.
4. The client SW component (and all other connected clients):
    a. Receives the BITE fault summary message via the connected SUB socket, providing comprehensive status of each known BITE fault. This message type can be ignored by other clients with established connections.
    b. Continues to receive periodic message updates sent in response to detected changes in fault status via the connected SUB socket.

Message Definitions

BITE Fault

The *BITEFault* message allows reporting a single detected hardware or software BITE fault or event which has occurred on a given LRU. When a change in fault state is detected, a message is sent which includes information about which LRU the fault occurred at, the category of fault, the application name (for software faults), an assigned fault index number (unique within each category or application name), and the current state of the fault. Faults which are no longer active can be cleared by transmitting sending a message with the *fault_status* set to CLEAR. Some BITE faults have no state, such as an indication that an LRU reset has occurred, and these are known as events and can be reported using the *fault_status* of UNKNOWN.

ZMQ Frame No.	Data

0	`BITEFault`
1	MsgHeader

2	Google Protocol Buffers message data in binary format, using the following message definition:
	`message BITEFault`
	`{`
	`enum Category`
	`{`
	`GENERIC = 1;`
	`POST = 2;`
	`HARDWARE = 3;`
	`SOFTWARE = 4;`
	`POWER = 5;`
	`}`
	`enum State`
	`{`
	`UNKNOWN = 1; // Fault status is unknown, or not applicable (used for reporting stateless events)`
	`CLEAR = 2; // Fault is no longer active, and should be cleared`
	`ACTIVE = 3; // Fault is raised`
	`}`
	`// NOTE: Field numbers 1-5 allow for compatibility with the existing Avant IFE BITE protobuf definition`

```
// Fields defining the specific
BITE fault
required string lru_name = 1; //
Hostname or unique identifier of
the LRU on which the fault
occurred
optional Category fault_category =
100 [default = SOFTWARE]; //
General category of fault
required string application_name =
2; // Application name (limited to
30 characters), used for reporting
software faults (not applicable
for other categories)
required sint32 fault_index = 3;
// Numeric index of the fault
(value range is 01..99), unique
within each category or SW
application
required string fault_parameter =
5; // Optional fault parameter
string, used to differentiate
between otherwise identical faults
```

optional string sub_system = 103; // Optional string indicating the sub-system this fault relates to

```
// BITE fault metadata
optional sint64 fault_timestamp =
101; // Optional timestamp of the
fault; specified in seconds since
epoch time (1970-01-01T00:00:00Z)

// BITE fault data
required State fault_status = 4;
// Current status of the fault
optional string fault_short_desc =
102; // Optional short description
of the fault (can be used to
provide additional detail when
reporting a stateless event)
}
```

BITE Fault Summary

The *BITEFaultSummary* message, transmitted on client subscription activity (during the initial connection, or in the case of a re-connection) and then at a configurable periodic interval, consists of a series of *BITEFa ult* messages which represent the complete state of all known BITE faults. Specifically, each unique BITE fault which has a **known state** should be included in the *BITEFaultSummary* message to present subscribing clients with an accurate view of the system (including any BITE faults which may have occurred while the SUB socket was disconnected).

ZMQ Frame No.	Data
0	`BITEFaultSummary`
1	MsgHeader

2	Google Protocol Buffers message data in binary format, using the following message definition:
	`message BITEFaultSummary`
	`{`
	`required sint64 lastFaultTimestamp = 1; // Timestamp of the most recent fault; specified in seconds since epoch time (1970-01-01T00:00:00Z)`
	`repeated BITEFault BITEFaults = 2; // Summary of known BITE faults, and their current status`
	`}`

BITE Faults

The following table defines the different BITE faults that may be published from the Connectivity VM

Sub-System	LRU Name	Category	Application Name	Fault Index	Description
MPS	MPS	Software	ConnectivityVM	1	Connectivity services are not available due to a failure in one or more sub-systems. There should be a more specific BITE fault that goes along with this (i.e. GX ModMan comms failure).
MPS	MPS	Software	ConnectivityVM	2	Could not connect to GCMS ground services even though a satellite connection is available.

MPS	MPS	Software	ConnectivityVM	3	Error receiving flight data from IFE VM.
MPS	MPS	Software	PortalVM	1	There is a communication failure with the Portal VM.
MPS	MPS	Software	PortalVM	2	The Portal VM is not running.
MAP	MPS	Software	MachineInstallation	1	There has been a communication failure with the MPS MAP Machine Installation API.
MAP	MPS	Software	GuestMachine	2	There has been a communication failure with the MPS MAP Guest Machine Control and Status API.
MAP	MPS	Software	MapStatus	3	There has been a communication failure with the MPS MAP Status API.
MAP	MPS	Software	MachineAlerts	4	There has been a communication failure with the MPS MAP Machine Alerts API.
MAP	MPS	Software	NetworkServices	5	There has been a communication failure with the MPS MAP Network Services API.

MAP	MPS	Software	HostDomain	6	There has been a communication failure with the MPS MAP Host Domain API.
MAP	MPS	Software	SecureStorage	7	There has been a communication failure with the MPS MAP Secure Storage API.
MAP	MPS	Software	TpmService	8	There has been a communication failure with the MPS MAP Trusted Platform Management Service API.
MAP	MPS	Software	SqlService	9	There has been a communication failure with the MPS MAP SQL Service API.
MAP	MPS	Software	ArincService	10	There has been a communication failure with the MPS MAP ARINC Service API.
MPS	MPS	Hardware	N/A	1	The MPS has reported a failure of one or more SSDs.
MPS	MPS	Hardware	N/A	2	The MPS has reported a temperature fault.
MPS	MPS	Hardware	N/A	3	The MPS has reported a power fault.

MPS	MPS	Hardware	N/A	4	The MPS has reported a volatile memory fault.
MPS	MPS	Hardware	N/A	5	The MPS has reported a failure with the internal MPS switch.
WAP	WAPm	Hardware	N/A	1	Communication failure with WAP m.
WAP	WAPm	Hardware	N/A	2	One or more radios in WAP m have a fault.
WAP	WAPm	Hardware	N/A	3	WAP m has exceeded the configured threshold for transmit errors.
WAP	WAPm	Hardware	N/A	4	WAP m has exceeded the configured threshold for receive errors.
WAP	WAPm	Hardware	N/A	5	WAP m has an internal fault.
WAP	WAPm	Hardware	N/A	6	WAP m has detected a power failure.
SatCom	SatCom	Generic	N/A	1	The SatCom system is in a fault state and is not functional.
SatCom	SatCom	Generic	N/A	2	There is a communications fault with the SatCom system.

SatCom	SatCom	Hardware	N/A	1	The SatCom system has a fault with receiving or parsing incoming ARINC data.
SatCom	SatCom	Hardware	N/A	2	The SatCom system has a fault with receiving or parsing incoming GPS data.
SatCom (GX)	ModMan	Hardware	N/A	1	There is a communications fault with the GX ModMan LRU.
SatCom (GX)	ModMan	Hardware	N/A	2	There is a temperature fault with the GX ModMan LRU.
SatCom (GX)	ModMan	Hardware	N/A	3	There is an internal fault with the GX ModMan LRU.
SatCom (GX)	ModMan	Hardware	N/A	4	The GX ModMan LRU has detected a power failure.
SatCom (GX)	KRFU	Hardware	N/A	1	There is a communication fault with the GX KRFU LRU.
SatCom (GX)	KRFU	Hardware	N/A	2	There is a temperature fault with the GX KRFU LRU.

SatCom (GX)	KRFU	Hardware	N/A	3	There is an internal fault with the GX KRFU LRU.
SatCom (GX)	KRFU	Hardware	N/A	4	The GX KRFU LRU has detected a power failure.
SatCom (GX)	KANDU	Hardware	N/A	1	There is a communicatio ns fault with the GX KANDU LRU.
SatCom (GX)	KANDU	Hardware	N/A	2	There is a temperature fault with the GX KANDU LRU.
SatCom (GX)	KANDU	Hardware	N/A	3	There is an internal fault with the GX KANDU LRU.
SatCom (GX)	KANDU	Hardware	N/A	4	The GX KANDU LRU has detected a power failure.
SatCom (GX)	OAE	Hardware	N/A	1	There is a communicatio ns fault with the GX OAE LRU.
SatCom (GX)	OAE	Hardware	N/A	2	There is a temperature fault with the GX OAE LRU.
SatCom (GX)	OAE	Hardware	N/A	3	There is an internal fault with the GX OAE LRU.
SatCom (GX)	OAE	Hardware	N/A	4	The GX OAE LRU has detected a power failure.

SatCom (GX)	APM	Hardware	N/A	1	There is a communications fault with the GX APM LRU.
SatCom (GX)	APM	Hardware	N/A	2	There is an internal fault with the GX APM LRU.

Connectivity VM

The following is a list of interfaces and data which should be made available from the Connectivity VM for use by the Connectivity GUI and backend SW components.

SatCom System Control and Status Interfaces

This interface provides data to the IFE VM about the health and status of the satellite modem, antenna, and antenna controller. It also provides an interface to reset the SatCom system and to put it into a state for accepting a software upgrade.

SatCom Status Keys

The SatCom status messages are populated with a set of status keys and values. The list of values will be made up of a set of common SatCom values that are always present regardless of the SatCom provider along with other values which are SatCom provider specific.

SatCom Status Key Name	SatCom Status Key Description	Published Periodically
satcom.health_state	The current health status of the satcom system. Possible values are: normal - The satcom system is functioning normally failed - The satcom system is not functioning unknown - The status of the satcom system is still being determined resetting - The satcom system is currently resetting.	yes
satcom.link_state	The current state of the satellite connection. Possible values are: online, offline	yes
satcom.handover_pending	"true" if a handover is pending, otherwise "false"	yes

satcom.tx_muted	"true" if the satcom system is currently muted from transmitting. "false" if it is not currently muted.	yes
satcom.tx_muted_reason	A string indicating the reason why the transmission from the satcom system is currently muted.	yes
satcom.signal_quality	The current quality of the satellite signal as a fractional Carrier to Noise signal ratio in 10 db/Hz format.	no
satcom.signal_level	The current power level in dBm of the satellite signal.	no
satcom.tx_byte_count	The number of bytes that have been transmitted through the satellite connection.	no
satcom.rx_byte_count	The number of bytes that have been received through the satellite connection.	no
satcom.link_start_time	The start time of the last satellite link in seconds since the epoch.	no
satcom.link_end_time	The end time of the last satellite link in seconds since the epoch.	no
satcom.firmware_version	The version of firmware loaded on the satcom system.	no
satcom.config_version	The version of configuration loaded on the satcom system.	no
satcom.serial_nbr	The serial number of the satcom system installed on the aircraft.	no
satcom.aircraft.latitude	The current latitude determined by the satcom system in arc seconds (1/3600 of a degree, from -324,000 at the south pole to +324,000 at the north pole, or -999,999 if not available).	no

satcom.aircraft.longitude	The current longitude determined by the satcom system in arc seconds (1/3600 of a degree, from -324,000 at the south pole to +324,000 at the north pole, or -999,999 if not available)..	no
satcom.aircraft.altitude	The current altitude determined by the satcom system in feet (+100,000 to -10,000, or -999,999 if not available).	no
satcom.aircraft.true_heading	The current true heading determined by the satcom system in 10 degrees format (0 (north) to 3,599, 9,999 if not available).	no
satcom.aircraft.ground_speed	The current ground speed determined by the satcom system in knots (0 to 1500, 9999 if not available).	no
satcom.aircraft.tail_nbr	The current aircraft tail number as a string.	no
satcom.aircraft.on_ground	"true" if the satcom system determines that the aircraft is on the ground, otherwise "false".	no
satcom.aircraft.pitch_angle	The current pitch angle determined by the satcom system in 10 degrees format (-900 (pitch down) to +900 (pitch up), -999,999 if not available).	no
satcom.aircraft.roll_angle	The current roll angle determined by the satcom system in 10 degrees format (-900 (roll left) to +900 (roll right), -999,999 if not available).	no
satcom.gx.modman.status	A GX specific value indicating the current health status of the Modem Manager. Possible values are: normal, warning, failed, resetting.	GX
satcom.gx.krfu.status	A GX specific value indicating the current health status of the Ku/Ka Radio Frequency Unit. Possible values are: normal, warning, failed, resetting.	GX

satcom.gx.kandu.status	A GX specific value indicating the current health status of the Ku/Ka Aircraft Data Network Unit. Possible values are: normal, warning, failed, resetting.	GX
satcom.gx.oae.status	A GX specific value indicating the current health status of the Outside Antenna Equipment. Possible values are: normal, warning, failed, resetting.	GX
satcom.gx.irs.status	A GX specific value indicating the current health status of the Inertial Reference System. Possible values are: normal, warning, failed, resetting.	no
satcom.gx.modman.oem_hw_pn	A GX specific value indicating the Modem Manager OEM hardware part number as a string.	no
satcom.gx.modman.sw_pn_valid	A GX specific value indicating if the Modem Manager software part number is valid. "true" if valid, otherwise "false".	no
satcom.gx.oae.oem_hw_pn	A GX specific value indicating the Outside Antenna Equipment OEM hardware part number as a string.	no
satcom.gx.oae.sw_pn_valid	A GX specific value indicating if the Outside Antenna Equipment software part number is valid. "true" if valid, otherwise "false".	no
satcom.gx.kandu.oem_hw_pn	A GX specific value indicating the Ku/Ka Aircraft Data Network Unit OEM hardware part number as a string.	no
satcom.gx.kandu.sw_pn_valid	A GX specific value indicating if the Ku/Ka Aircraft Data Network Unit software part number is valid. "true" if valid, otherwise "false".	no
satcom.gx.krfu.oem_hw_pn	A GX specific value indicating the Ku/Ka Radio Frequency Unit OEM hardware part number as a string.	no

satcom.gx.krfu.sw_pn_valid	A GX specific value indicating if the Ku/Ka Radio Frequency Unit software part number is valid. "true" if valid, otherwise "false".	no
satcom.gx.apm.oem_hw_pn	A GX specific value indicating the Airplane Personality Module OEM hardware part number as a string.	no
satcom.gx.apm.sw_pn_valid	A GX specific value indicating if the Airplane Personality Module software part number is valid. "true" if valid, otherwise "false".	no
satcom.gx.modman.lru_name	A GX specific value indicating the name of the Modem Manager LRU as a string.	no
satcom.gx.modman.sn	A GX specific value indicating the serial number of the Modem Manager as a string.	no
satcom.gx.modman.sw_subpart_ver	A GX specific value indicating the software sub-part version number for the Modem Manager as a string. Formatted as"SW=CRC;SW=CRC;SW=CRC". Any number of SW/CRC pairs to the limit of the string may be formatted. CRC and SW is in a proprietary format.	no
satcom.gx.modman.oem_id	A GX specific value indicating the OEM id of the Modem Manager as a string.	no
satcom.gx.modman.honeywell_hw_pn	A GX specific value indicating the Honeywell hardware part number for the Modem Manager as a string.	no
satcom.gx.modman.subassembly_id	A GX specific value indicating the sub-assembly id for the Modem Manager as a string.	no
satcom.gx.kandu.lru_name	A GX specific value indicating the name of the Ku/Ka Aircraft Data Network Unit LRU as a string.	no
satcom.gx.kandu.sn	A GX specific value indicating the serial number of the Ku/Ka Aircraft Data Network Unit as a string.	no

satcom.gx.kandu.sw_subpart_ver	A GX specific value indicating the software sub-part version number for the Ku/Ka Aircraft Data Network Unit as a string. Formatted as"SW=CRC;SW=CRC;SW=CRC". Any number of SW/CRC pairs to the limit of the string may be formatted. CRC and SW is in a proprietary format.	no
satcom.gx.kandu.oem_id	A GX specific value indicating the OEM id of the Ku/Ka Aircraft Data Network Unit as a string.	no
satcom.gx.kandu.honeywell_hw_pn	A GX specific value indicating the Honeywell hardware part number for the Ku/Ka Aircraft Data Network Unit as a string.	no
satcom.gx.kandu.subassembly_id	A GX specific value indicating the sub-assembly id for the Ku/Ka Aircraft Data Network Unit as a string.	no
satcom.gx.krfu.lru_name	A GX specific value indicating the name of the Ku/Ka Radio Frequency Unit LRU as a string.	no
satcom.gx.krfu.sn	A GX specific value indicating the serial number of the Ku/Ka Radio Frequency Unit as a string.	no
satcom.gx.krfu.sw_subpart_ver	A GX specific value indicating the software sub-part version number for the Ku/Ka Radio Frequency Unit as a string. Formatted as"SW=CRC;SW=CRC;SW=CRC". Any number of SW/CRC pairs to the limit of the string may be formatted. CRC and SW is in a proprietary format.	no
satcom.gx.krfu.oem_id	A GX specific value indicating the OEM id of the Ku/Ka Radio Frequency Unit as a string.	no
satcom.gx.krfu.honeywell_hw_pn	A GX specific value indicating the Honeywell hardware part number for the Ku/Ka Radio Frequency Unit as a string.	no

satcom.gx.krfu.subassembly_id	A GX specific value indicating the sub-assembly id for the Ku/Ka Radio Frequency Unit as a string.	no
satcom.gx.oae.lru_name	A GX specific value indicating the name of the Outside Antenna Equipment LRU as a string.	no
satcom.gx.oae.sn	A GX specific value indicating the serial number of the Outside Antenna Equipment as a string.	no
satcom.gx.oae.sw_subpart_ver	A GX specific value indicating the software sub-part version number for the Outside Antenna Equipment as a string. Formatted as"SW=CRC;SW=CRC;SW=CRC". Any number of SW/CRC pairs to the limit of the string may be formatted. CRC and SW is in a proprietary format.	no
satcom.gx.oae.oem_id	A GX specific value indicating the OEM id of the Outside Antenna Equipment as a string.	no
satcom.gx.oae.honeywell_hw_pn	A GX specific value indicating the Honeywell hardware part number for the Outside Antenna Equipment as a string.	no
satcom.gx.oae.subassembly_id	A GX specific value indicating the sub-assembly id for the Outside Antenna Equipment as a string.	no
satcom.gx.apm.lru_name	A GX specific value indicating the name of the Aircraft Personality Module LRU as a string.	no
satcom.gx.apm.sn	A GX specific value indicating the serial number of the Aircraft Personality Module as a string.	no
satcom.gx.apm.oem_id	A GX specific value indicating the OEM id of the Aircraft Personality Module as a string.	no

SatCom Error Codes

The following table lists the possible error codes that can be returned from the SatCom interfaces in addition to the common error codes:

Error Name	Error Code	Error Description
SATCOM_COMM_FAILURE	1000	There was an error communicating with the SatCom system.

Periodic SatCom Status

A set of modem status values that are sent at a configurable periodic rate. The format of the payload section for this message is the following GPB message:

Message Name: SatComStatus

Fields:

Name	Status	Type	Description	Example
status_values	REPEATED	Property	The array of status values.	{"satcom.common.link_state","ONLINE"}

IDL:

```
message SatComStatus
{
   repeated Property status_values = 1;
}
```

Usage Summary

Design Pattern	ZMQ XPUB/SUB
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of XPUB socket on well-known 192.168.10.254:49120 (with *ZMQ_XPUB_VERBOSE* sockopt)
Client Socket(s)	*connect()* of SUB socket using an ephemeral port (from client such as IFE VM)

<table>
<tr><td>Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td>SatComStatus</td></tr>
<tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>SatComStatus</td></tr>
</table></td></tr>
</table>

Request SatCom Status

Allows the IFE VM to request one or more specific SatCom status values. The format of the request message is:

Message Name: SatComStatusReq

Fields:

Name	Status	Type	Description	Example
status_keys	REPEATED	string	The array of status keys that the IFE VM is requesting.	["satcom.common.link_status","satcom.gx.update_mode"]

IDL:

```
message SatComStatusReq
{
   repeated string status_keys = 1;
}
```

The response message has the following format:

Message Name: SatComStatusResp

Fields:

Name	Status	Type	Description	Example

fields	REPEATED	SatComStatusValue	A structure containing a Property object indicating the key name and value, and an optional ErrorMsg if there was an error retrieving the status for the specified key.	

IDL:

```
message SatComStatusResp
{
   message SatComStatusValue
   {
      required Property value = 1;
      optional ErrorMsg error = 2;
   }

   repeated SatComStatusValue status_values = 1;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of REP socket on well-known 192.168.10.254:49121 (with ZMQ_REP sockopt)
Client Socket(s)	*connect()* of REQ socket using an ephemeral port (from client such as IFE VM)

<table>
<tr><td>Request Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>SatComStatusReq</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>SatComStatusReq</td></tr></table></td></tr>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>SatComStatusResp</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>SatComStatusResp</td></tr></table></td></tr>
</table>

Set SatCom Download Mode

Provides an interface for setting the download mode. The request message has the following format:

Message Name: SetSatComDownloadModeReq

Fields:

Name	Status	Type	Description	Example
enabled	REQUIRED	bool	Specifies if the download mode should be enabled (true) or disabled (false).	true

IDL:

```
message SetSatComDownloadModeReq
{
   required bool enabled = 1;
}
```

The response message has the following format:

Message Name: SetSatComDownloadModeResp

Fields:

Name	Status	Type	Description	Example
success	REQUIRED	bool	Returns true if successful, false for failure.	true
error	OPTIONAL	ErrorMsg	Optional error code and description if the request failed.	{1,"Internal error"}

IDL:

```
message SetSatComDownloadModeResp
{
   required bool success = 1;
   optional ErrorMsg error = 2;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of REP socket on well-known 192.168.10.254:49121 (with ZMQ_REP sockopt)
Client Socket(s)	*connect()* of REQ socket using an ephemeral port (from client such as IFE VM)

<table>
<tr><td>Request Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>SetSatComDownloadModeReq</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>SetSatComDownloadModeReq</td></tr></table></td></tr>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>SetSatComDownloadModeResp</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>SetSatComDownloadModeResp</td></tr></table></td></tr>
</table>

Reset SatCom

Provides an interface for resetting the SatCom system. There is no payload message associated with this request. Just the request name (SatComResetReq) along with the common message header. The response message has the following format:

Message Name: SatComResetResp

Fields:

Name	Status	Type	Description	Example
success	REQUIRED	bool	Returns true if successful, false for failure.	true
error	OPTIONAL	ErrorMsg	Optional error code and description if the request failed.	{1,"Internal error"}

IDL:

```
message SatComResetResp
{
   required bool success = 1;
   optional ErrorMsg error = 2;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of REP socket on well-known 192.168.10.254:49121 (with ZMQ_REP sockopt)
Client Socket(s)	*connect()* of REQ socket using an ephemeral port (from client such as IFE VM)

<table>
<tr><td>Request Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>SatComReset Req</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>None</td></tr></table></td></tr>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>SatComReset Resp</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>SatComReset Resp</td></tr></table></td></tr>
</table>

WAP Control and Status Interface

This interface provides data to the IFE VM about the health and status of the WAPs connected to the system. It also provides interfaces for controlling WAP radios and resetting WAPs.

WAP Status Keys

The WAP status messages are populated with a set of status keys and values. The list of values will be made up of a set of common WAP values that are always present regardless of the types of WAPs installed along with other values which are WAP type specific. Individual WAP's are designated by a number 1-9 or a * to designate all WAPs. Radios are designated by the radio number or * for all radios.

WAP Status Key Name	WAP Status Key Description	Published Periodically
wap.count	An integer value indicating the number of WAPs configured in the system. This is based off of a configuration and not on the number of WAPs actually communicating at any period of time.	yes
wap.m.status	The status of the WAP indicated by "m". Possible values are "ok", "failed", "unknown", "resetting"	yes
wap.m.radio_count	The number of radios in the WAP indicated by "m".	yes
wap.m.firmware_version	The version of the firmware currently loaded on the WAP indicated by "m".	no
wap.m.config_version	The version of the configuration currently loaded on the WAP indicated by "m".	no
wap.m.serial_nbr	The serial number of the WAP indicated by "m".	no
wap.m.product_name	The product name of the WAP indicated by "m".	no
wap.m.up_time	The amount of time in seconds that the WAP specified by "m" has been running since its last reboot.	no
wap.m.tx_packet_count	The total number of packets transmitted by all the radios in the WAP specified by "m".	no
wap.m.rx_packet_count	The total number of packets received by all the radios in the WAP specified by "m".	no
wap.m.tx_byte_count	The total number of bytes transmitted by all the radios in the WAP specified by "m".	no
wap.m.rx_byte_count	The total number of bytes received by all the radios in the WAP specified by "m".	no

wap.m.country_code	The current country code of the country the WAP believes it's located in.	no
wap.m.radio.n.status	The current state of radio "n" on WAP "m". The list of possible strings match the RadioState types from the next section.	yes
wap.m.radio.n.frequency_band	The frequency of radio "n" on WAP "m". The possible values are "2.4" or "5".	yes
wap.m.radio.n.channel	The RF channel of radio "n" on WAP "m". The different channels available for each of the frequency bands are available at https://en.wikipedia.org/wiki/List_of_WLAN_channels.	yes
wap.m.radio.n.max_power	"true" if radio "n" on WAP "m" is currently set to maximum power. Otherwise "false".	yes
wap.m.radio.n.curr_power	The current power in dBm of the radio "n" on WAP "m".	no
wap.m.radio.n.mac_address	The MAC address of radio "n" on WAP "m".	no
wap.m.radio.n.tx_unicast_packet_count	The number of unicast packets transmitted from radio "n" on WAP "m".	no
wap.m.radio.n.tx_packet_error_count	The number of packets that had an error in transmission on radio "n" on WAP "m".	no
wap.m.radio.n.tx_packet_count	The number of packets transmitted from radio "n" on WAP "m".	no
wap.m.radio.n.rx_packet_count	The number of packets received on radio "n" on WAP "m".	no
wap.m.radio.n.tx_byte_count	The number of bytes transmitted from radio "n" on WAP "m".	no
wap.m.radio.n.rx_byte_count	The number of bytes transmitted from radio "n" on WAP "m".	no
wap.m.radio.n.vap_count	The number of virtual access points configured for radio "n" on WAP "m".	no

wap.m.radio.n.vap.x.ssid	The virtual access point (VAP) SSID for VAP "x" on radio "n" on WAP "m".	no
wap.m.radio.n.vap.x.broadcast_ssid	A boolean value indicating if the SSID for VAP "x" on radio "n on WAP "m" is a broadcast SSID. "true" if it is, otherwise "false".	no
wap.m.radio.n.vap.x.user_count	The number of users associated on VAP "x" on radio "n" on WAP "m".	no

Common WAP Data Types

The following common WAP data types are used throughout the WAP Status and Control Interfaces.

Name	Type	Description
WapId	Enumeration	List of unique identfiers for all of the different WAPs in the system. Contains the following values: ALL, WAP1, WAP2, WAP3, WAP4, WAP5, WAP6, WAP7, WAP8, WAP9
RadioType	Enumeration	The different types of WAP radios in the system. Possible values are: ALL, 2_4GHz, 5GHz
RadioStatus	Enumeration	The different possible states for each of the WAP radios. Possible values are: DEFAULT, ENABLED, DISABLED, UNKNOWN

WapOpResp

This structure is used to indicate if a requested WAP operation succeeded or failed and if it failed optionally provide an error code and error description.

Fields:

Name	Status	Type	Description	Example
success	REQUIRED	boolean	true if the operation was successful, otherwise false.	true
wap_id	OPTIONAL	WapId	The ID of the WAP that this belongs to.	WAP1

radio_type	OPTIONAL	RadioType	The type of the radio that belongs to.	5GHz
error	OPTIONAL	ErrorMsg	Optional error code and description if the request failed.	{1,"Internal error"}

IDL:

```
message WapOpResp
{
   required boolean success = 1;
   optional ErrorMsg error = 2;
}
```

WAP Error Codes

The following table contains the error codes and descriptions for errors on the WAP interfaces:

Error Name	Error Code	Error Description
WAP_COMM_FAILURE	2000	There was an error communicating with the specified WAP.
INVALID_WAP	2001	The specified WAP does not exist.
INVALID_RADIO	2002	The specified WAP radio does not exist.
INVALID_VAP	2003	The specified Virtual Access Point does not exist.

Periodic WAP Status

A set of WAP status values that are sent at a configurable periodic rate. The format of this message is:

Message Name: WapStatus

Fields:

Name	Status	Type	Description	Example
status_values	REPEATED	Property	The array of status values.	{"wap.wap1.common.radios.1.state","ENABLED"}

IDL:

```
message WapStatus
{
   repeated Property status_values = 1;
```

}

Usage Summary

Design Pattern	ZMQ XPUB/SUB
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of XPUB socket on well-known 192.168.10.254:49130 (with *ZMQ_XPUB_VERBOSE* sockopt)
Client Socket(s)	*connect()* of SUB socket using an ephemeral port (from client such as IFE VM)
Message Format	Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data

ZMQ Frame No.	Description	Example
0	Message Name (string)	WapStatus
1	Message Header (Google Protocol Buffers binary format)	MsgHeader
2	Message Payload (Google Protocol Buffers binary format)	WapStatus

Request WAP Status

Allows the IFE VM to request one or more specific WAP status values. The format of the request message is:

Message Name: WapStatusReq

Fields:

Name	Status	Type	Description	Example
status_keys	REPEATED	string	The array of status keys that the IFE VM is requesting.	["wap.wap1.common.radios.1.state","wap.wap1.common.radios.2.state"]

IDL:

```
message WapStatusReq
{
   repeated string status_keys = 1;
}
```

The response message has the following format:

Message Name: WapStatusResp

Fields:

Name	Status	Type	Description	Example
fields	REPEATED	WapStatusValue	A structure containing a Property object indicating the key name and value, and an optional ErrorMsg if there was an error retrieving the status for the specified key.	

IDL:

```
message WapStatusResp
{
   message WapStatusValue
   {
      required Property value = 1;
      optional ErrorMsg error = 2;
   }

   repeated WapStatusValue status_values = 1;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of REP socket on well-known 192.168.10.254:49131 (with ZMQ_REP sockopt)
Client Socket(s)	*connect()* of REQ socket using an ephemeral port (from client such as IFE VM)

<table>
<tr><td>Request Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>WapStatusReq</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>WapStatusReq</td></tr></table></td></tr>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>WapStatusResp</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>WapStatusResp</td></tr></table></td></tr>
</table>

Set WAP Radio State

Provides an interface for enabling or disabling one or more radios on one or more WAPs. The request message has the following format:

Message Name: WapSetRadioStateReq

Fields:

Name	Status	Type	Description	Example
setRadioState	REPEATED	SetRadioState	A list of SetRadioState messages specifying the state	[{WAP1, ALL, DEFAULT}]

IDL:

```
message WapSetRadioStateReq
{
   repeated SetRadioState set_radio_state = 1;
}
```

The SetRadioState message has the following structure:

Message Name: SetRadioState

Fields:

Name	Status	Type	Description	Default	Example
id	OPTIONAL	WapId	The ID of the WAP that this message applies to.	ALL	ALL
type	OPTIONAL	RadioType	The type of the radio.	ALL	ALL
state	OPTIONAL	RadioState	The state the WAP radio should be set to.	DEFAULT	DEFAULT

IDL:

```
message SetRadioState
{
   optional WapId id = 1;
   optional RadioType type = 2;
   optional RadioState state = 3;
}
```

The response message has the following format:

Message Name: WapSetRadioStateResp

Fields:

Name	Status	Type	Description	Example

results	REPEATED	WapOpResp	An array of WapOpResp objects indicating the result of the operation on each specified WAP.	[{true},{false,{1," Internal Error"}}]

IDL:

```
message WapSetRadioStateResp
{
   repeated WapOpResp results = 1;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of REP socket on well-known 192.168.10.254:49131 (with ZMQ_REP sockopt)
Client Socket(s)	*connect()* of REQ socket using an ephemeral port (from client such as IFE VM)
Request Message Format	Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data

ZMQ Frame No.	Description	Example
0	Message Name (string)	WapSetRadio StateReq
1	Message Header (Google Protocol Buffers binary format)	MsgHeader
2	Message Payload (Google Protocol Buffers binary format)	WapSetRadio StateReq

<table>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td>WapSetRadioStateResp</td></tr>
<tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>WapSetRadioStateResp</td></tr>
</table></td></tr>
</table>

Set WAP Radio Max Power

Provides an interface for setting one or more radios on one or more WAPs to maximum power. The request message has the following format

Message Name: WapSetRadioMaxPowerReq

Fields:

Name	Status	Type	Description	Example
setRadioMaxPower	REPEATED	SetRadioMaxPower	A list of SetRadioMaxPower messages specifying the state	true

IDL:

```
message WapSetRadioMaxPowerReq
{
   repeated SetRadioMaxPower set_radio_max_power = 1;
}
```

The SetRadioMaxPower message has the following structure:

Message Name: SetRadioMaxPower

Fields:

Name	Status	Type	Description	Default

id	OPTIONAL	WapId	The ID of the WAP that this message applies to.	ALL
type	OPTIONAL	RadioType	The type of the radio.	ALL
max_power	OPTIONAL	bool	True if the radio should be set to max power, false if the radio should be set to it's configured power level.	true

IDL:

```
message SetRadioMaxPower
{
   optional WapId id = 1;
   optional RadioType type = 2;
   optional bool max_power = 3;
}
```

The response message has the following format:

Message Name: WapSetRadioStateResp

Fields:

Name	Status	Type	Description	Example
results	REPEATED	WapOpResp	An array of WapOpResp objects indicating the result of the operation on each specified WAP.	[{true},{false,{1," Internal Error"}}]

IDL:

```
message WapSetRadioStateResp
{
   repeated WapOpResp results = 1;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of REP socket on well-known 192.168.10.254:49131 (with ZMQ_REP sockopt)

<table>
<tr><td>Client Socket(s)</td><td>connect() of REQ socket using an ephemeral port (from client such as IFE VM)</td></tr>
<tr><td>Request Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data
<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td>WapSetRadioMaxPowerReq</td></tr>
<tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>WapSetRadioMaxPowerReq</td></tr>
</table></td></tr>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data
<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td>WapSetRadioMaxPowerResp</td></tr>
<tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>WapSetRadioMaxPowerResp</td></tr>
</table></td></tr>
</table>

Reset WAP

Provides an interface for resetting one or more WAPs. The request message has the following format:

Message Name: WapResetReq

Fields:

Name	Status	Type	Description	Example
id	REPEATED	WapId	The ID of the WAP to reset or ALL	Wap1

IDL:

```
message WapResetReq
{
   repeated WapId id = 1;
}
```

The response message has the following format:

Message Name: WapResetResp

Fields:

Name	Status	Type	Description	Example
results	REPEATED	WapOpResp	An array of WapOpResp objects indicating the result of the operation on each specified WAP.	[{true},{false,{1," Internal Error"}}]

IDL:

```
message WapResetResp
{
   repeated WapOpResp results = 1;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of REP socket on well-known 192.168.10.254:49131 (with ZMQ_REP sockopt)
Client Socket(s)	*connect()* of REQ socket using an ephemeral port (from client such as IFE VM)

<table>
<tr><td>Request Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>WapResetReq</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>WapResetReq</td></tr></table></td></tr>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>WapResetResp</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>WapResetResp</td></tr></table></td></tr>
</table>

User Internet Service

The following interfaces provide status and control of user internet service.

Common User Internet Service Data Types

The following common User Internet Service data types are used throughout this status and control

Interfaces.

Name	Type	Description
ServiceState	Enumeration	List of unqiue states for User Internet Service. Contains the following values: AVAILABLE - Internet service is available RESTRICTED - Internet service is denied to users due to the flight state UNAVAILABLE - Internet service is currently not functionally available DISABLED - Internet service has been manually disabled UNKNOWN - Internet service availability has not yet been determined.
OverrideStatus	Enumeration	List of unique status for the current user internet service override. Contains the following values: OVERRIDDEN - User internet service has been overridden. NOT_OVERRIDDEN - User internet service has not been overridden. OVERRIDE_NOT_POSSIBLE - It is not currently possible to override internet service.

UserInetSvcDetailStatusCode	Enumeration	List of status codes for user internet service providing more detailed status information than the state. Contains the following values: OUT_OF_COVERAGE - Internet service is unavailable because the system is out of the coverage area. MODEM_COMMS_ERROR - There is a problem communicating with the satellite modem. GROUND_SERVICES_COMMS_ERROR - There is a problem communicating with ground services that is affecting user internet service.

Periodic User Internet Status

The following message will be published to subscribers at a configurable periodic rate to indicate the current state of internet service for each user group in the system. The format of this message is:

Message Name: UserInetSvcStatus

Fields:

Name	Status	Type	Description	Example
svcStatus	REPEATED	GroupInternetServiceStatus	List of user internet status values (one per configured group).	[{"passenger",AVAILABLE}]

IDL:

```
message UserInetSvcStatus
{
   repeated GroupInternetServiceStatus svc_status = 1;
}
```

The format of the GroupInternetServiceStatus message is:

Name	Status	Type	Description	Example
group	required	string	The name of the user group this status applies to.	"passenger"

state	required	ServiceState	The current state of user internet service for the specified group.	AVAILABLE
override_status	required	OverrideStatus	The current override status for the specified user group.	OVERRIDDEN
status_code	optional	UserInetSvcDetailStatusCode	A status code with a more detailed indication of the current state of the service.	OUT_OF_COVERAGE

IDL:

```
message GroupInternetServiceStatus
{
   required string group = 1;
   required ServiceState state = 2;
   required OverrideStatus override_status = 3;
   optional UserInetSvcDetailStatusCode status_code = 4;
}
```

Usage Summary

Design Pattern	ZMQ XPUB/SUB
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of XPUB socket on well-known 192.168.10.254:49140 (with *ZMQ_XPUB_VERBOSE* sockopt)
Client Socket(s)	*connect()* of SUB socket using an ephemeral port (from client such as IFE VM)

<table>
<tr><td>Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td>UserInetSvcStatus</td></tr>
<tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>UserInetSvcStatus</td></tr>
</table></td></tr>
</table>

User Internet Service Override Request

This interface allows for the enabling of an internet service override. If the override is enabled then internet service will be made available to all users in the system. If it is not overridden then internet service will be made available to users based on the current flight state (i.e. geolocation, flight phase, etc). The format of the request message is:

Message Name: UserInetSvcOverrideReq

Fields:

Name	Status	Type	Description	Example
override_enabled	REQUIRED	bool	Specifies if the override should be enabled or disabled.	true

IDL:

```
message UserInetSvcOverrideReq
{
   required bool override_enabled = 1;
}
```

The response message has the following format:

Message Name: UserInetSvcOverrideResp

Fields:

Name	Status	Type	Description	Example

success	REQUIRED	bool	Returns true if successful, false for failure.	true
error	OPTIONAL	ErrorMsg	Optional error code and description if the request failed.	{1,"Internal error"}

IDL:

```
message UserInetSvcOverrideResp
{
   required bool success = 1;
   optional ErrorMsg error = 2;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of REP socket on well-known 192.168.10.254:49141 (with ZMQ_REP sockopt)
Client Socket(s)	*connect()* of REQ socket using an ephemeral port (from client such as IFE VM)
Request Message Format	Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data

ZMQ Frame No.	Description	Example
0	Message Name (string)	UserInetSvcOverrideReq
1	Message Header (Google Protocol Buffers binary format)	MsgHeader
2	Message Payload (Google Protocol Buffers binary format)	UserInetSvcOverrideReq

<table>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td>UserInetSvcOverrideResp</td></tr>
<tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>UserInetSvcOverrideResp</td></tr>
</table></td></tr>
</table>

IFE Internet Service

The following interface provides status and control for internet service to the IFE VM.

Common User Internet Service Data Types

The following common User Internet Service data types are used throughout this status and control Interfaces.

Name	Type	Description
IFEInetServiceState	Enumeration	List of unqiue states for User Internet Service. Contains the following values: AVAILABLE - Internet service is available UNAVAILABLE - Internet service is currently not functionally available UNKNOWN - Internet service availability has not yet been determined.

Periodic IFE Internet Status

The following message will be published to subscribers at a configurable periodic rate to indicate the current state of internet service for each user group in the system. The format of this message is:

Message Name: IFEInetSvcStatus

Fields:

Name	Status	Type	Description	Example
state	REQUIRED	IFEInetServiceState	The current state of internet service to the IFE VM.	AVAILABLE

IDL:

```
message IFEInetSvcStatus
{
   required IFEInetServiceState state = 1;
}
```

Usage Summary

Design Pattern	ZMQ XPUB/SUB
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of XPUB socket on well-known 192.168.10.254:49150 (with *ZMQ_XPUB_VERBOSE* sockopt)
Client Socket(s)	*connect()* of SUB socket using an ephemeral port (from client such as IFE VM)
Message Format	Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data

ZMQ Frame No.	Description	Example
0	Message Name (string)	IFEInetSvcStatus
1	Message Header (Google Protocol Buffers binary format)	MsgHeader
2	Message Payload (Google Protocol Buffers binary format)	IFEInetSvcStatus

Real Time Monitoring Service Interface

The Real Time Monitoring Service (RTMS) allows for the aircraft to provide real-time data to ground services. The service provides a key/value data store that the ground services can pull from as needed. This interface allows the IFE VM to add key/value pairs to that data store.

RTMS Data Submit Request

This interface allows the IFE VM to submit one or more key value pairs to the RTMS data store. If any key/value pair does not currently exist in the data store it will be added. If any key/value pair does already exist the vale will be updated to the new value in the message. The request message has the following format:

Message Name: RTMSDataSubmitReq

Fields:

Name	Status	Type	Description	Example
data_values	REPEATED	Property	The list of key/value pairs to be added to the RTMS data	[{"ife.states", "good"}]

IDL:

```
message RTMSDataSubmitReq
{
   repeated Property data_values = 1;
}
```

The response message has the following format:

Message Name: RTMSDataSubmitResp

Fields:

Name	Status	Type	Description	Example
success	REQUIRED	bool	Returns true if successful, false for failure.	true
error	OPTIONAL	ErrorMsg	Optional error code and description if the request failed.	{1,"Internal error"}

IDL:

```
message RTMSDataSubmitResp
{
   required bool success = 1;
   optional ErrorMsg error = 2;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse

<table>
<tr><td>Server Socket</td><td>Connectivity VM bind() of REP socket on well-known 192.168.10.254:49160 (with ZMQ_REP sockopt)</td></tr>
<tr><td>Client Socket(s)</td><td>connect() of REQ socket using an ephemeral port (from client such as IFE VM)</td></tr>
<tr><td>Request Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data
<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td>RTMSDataSubmitReq</td></tr>
<tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>RTMSDataSubmitReq</td></tr>
</table></td></tr>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data
<table>
<tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr>
<tr><td>0</td><td>Message Name (string)</td><td>RTMSDataSubmitResp</td></tr>
<tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr>
<tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>RTMSDataSubmitResp</td></tr>
</table></td></tr>
</table>

Log Transfer Interface

The Log Transfer Interface allows for the transferring of log files from the Connectivity VM to the IFE VM or vice versa. Depending on how the system is configured log files may be offloaded through the Connectivity VM via one of the wireless interfaces or through the IFE VM via USB connection to one of the DSU-D4s. This interface uses an Network File System (NFS) share provided by the Connectivity VM to perform the transfer of the log files between the VMs and a ZMQ/GPB interface for notification when log files are available.

NFS Share

The Connecitivity VM will make an NFS share available on the IP 192.168.10.254 at the /var/log/transfer mount point as a read/write share.

Transfer Logs Request

This interface allows one VM to request the transfer of log files to another. Both the IFE VM and the Connectivity VM will implement both sides of this interface. The request contains only the message header with the "TransferLogsReq" message name as there are no required parameters. When a request has been sent the VM receiving the request will transfer all logs from the NFS share to its own internal log storage area. The VM performing the transfer will delete each log file from the NFS share as each file is successfully transferred. The response message has the following format:

Message Name: TransferLogsResp

Fields:

Name	Status	Type	Description	Example
success	REQUIRED	bool	Returns true if successful, false for failure.	true
error	OPTIONAL	ErrorMsg	Optional error code and description if the request failed.	{1,"Internal error"}

IDL:

```
message TransferLogsResp
{
   required bool success = 1;
   optional ErrorMsg error = 2;
}
```

Usage Summary

Design Pattern	ZMQ REQ/REP
Security Pattern	Ironhouse
Server Socket	Connectivity VM *bind()* of REP socket on well-known 192.168.10.254:49170 (with ZMQ_REP sockopt), IFE VM bind() of REP socket on well-known 192.168.10.1:49170.

<table>
<tr><td>Client Socket(s)</td><td>connect() of REQ socket using an ephemeral port (from client such as IFE VM or Connectivity VM)</td></tr>
<tr><td>Request Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data
<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>TransferLogs Req</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr></table></td></tr>
<tr><td>Response Message Format</td><td>Multi-part, with dedicated frames for message name, metadata key/value pairs, and payload data
<table><tr><th>ZMQ Frame No.</th><th>Description</th><th>Example</th></tr><tr><td>0</td><td>Message Name (string)</td><td>TransferLogs Resp</td></tr><tr><td>1</td><td>Message Header (Google Protocol Buffers binary format)</td><td>MsgHeader</td></tr><tr><td>2</td><td>Message Payload (Google Protocol Buffers binary format)</td><td>TransferLogs Resp</td></tr></table></td></tr>
</table>