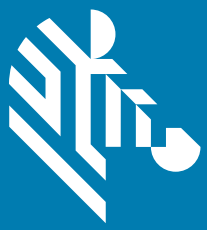
CLAS

Configuration and Location Analytics Software

API Developer Guide



ZEBRA

MN-003198-02

Copyright

Terms of Use

- Proprietary Statement

  This manual contains proprietary information of Zebra Technologies Corporation and its subsidiaries ("Zebra Technologies"). It is intended solely for the information and use of parties operating and maintaining the equipment described herein. Such proprietary information may not be used, reproduced, or disclosed to any other parties for any other purpose without the express, written permission of Zebra Technologies.
- Product Improvements

  Continuous improvement of products is a policy of Zebra Technologies. All specifications and designs are subject to change without notice.
- Liability Disclaimer

  Zebra Technologies takes steps to ensure that its published Engineering specifications and manuals are correct; however, errors do occur. Zebra Technologies reserves the right to correct any such errors and disclaims liability resulting therefrom.
- Limitation of Liability

  In no event shall Zebra Technologies or anyone else involved in the creation, production, or delivery of the accompanying product (including hardware and software) be liable for any damages whatsoever (including, without limitation, consequential damages including loss of business profits, business interruption, or loss of business information) arising out of the use of, the results of use of, or inability to use such product, even if Zebra Technologies has been advised of the possibility of such damages. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Revision History

Changes to the original guide are listed below:

Change	Date	Description
-01 Rev A	2/2019	Initial Rev A Release
-02 Rev A	02/2020	Updated the guide with RLTS, ATR7000, and CLAS information.

Table of Contents

About This Guide
- Introduction 5
- Chapter Descriptions 5
- Notational Conventions 5
- Related Documents 6
- Service Information 6
- Provide Documentation Feedback 6

ZAATS Introduction
- Introduction 7
- Product Overview 7
  - Description and Features 7
  - ATR7000 Advanced Array Reader (AAR) 8
  - RTLS Services 9
  - RTLS Configuration and Management (CNM) 9
  - Location Analytics (LA) 10
  - Radio Control & Data (CND) 10
  - ZAATS Interfaces 10

Configuring and Managing a ZAATS System
- Introduction 12
- Radio CND Configuration 12
  - Location Analytics (LA) Configuration 13

RTLS Rest API
- Introduction 16
- API Index 17
  - RTLS C&M REST API 17
  - ATR7000 Management REST API 18
  - Location Analytics REST API 19
  - CND Related REST API 20

API Programming Examples
- Introduction 21

Using the Python Requests HTTP Client to Call RTLS APIs 21
Getting Information About Various RTLS Components 22
Simple GET Operations 23
Compound GET Operations 23
Configuring RTLS System 25
Configuring Location and Event Endpoints 26
Multi-tag Item Configuration 26
Configuring Reference Tags 27
Customize LA Reporting 28
Set LA Filters 29
Monitoring and Managing RTLS System 30
Monitoring RTLS Health Status 30
Get All RTLS Errors and Warnings 31
Getting Latest RTLS Error and Warning Information 32
Reboot All Readers 32
Restart RTLS 32
Updating RTLS Components 33
Uploading Files to RTLS 33
Updating Reader Software 33

RTLS Location and Event Reporting
Messaging Platform Overview 36
Location Reporting 37
Message Formats 37
Example of a Location Update 38
Event Reporting 38
Message Formats 38
Field Description 39
Example of Event Reporting 39

About This Guide

Introduction

This guide provides programming information for the management and data interfaces that are required for application development for the Zebra Real-Time Location System (RTLS) system.

Chapter Descriptions

Topics covered in this guide are as follows:

- ZAATS Introduction provides an overview of Zebra's Advanced Asset Tracking System, including the ATR7000 overhead RFID readers and the RTLS Services software.
- Configuring and Managing a ZAATS System provides an overview of how to use the API.
- RTLS Rest API provides a listing of the REST URI and HTTP methods used to configure and manage ZAATS.
- API Programming Examples describes programming samples that utilize the RTLS Rest API.
- RTLS Location and Event Reporting provides an overview of the data interface that reports tag ID, location, and event information in ZAATS.

Notational Conventions

The following conventions are used in this document:

- The **Consolas** font is used throughout the guide to represent code.
- **Bold** text is used to highlight the following:
  - Dialog box, window and screen names
  - Drop-down list and list box names
  - Check box and radio button names
  - Icons on a screen
  - Key names on a keypad
  - Button names on a screen.

- Bullets (•) indicate:
  - Action items
  - Lists of alternatives
  - Lists of required steps that are not necessarily sequential.
- Sequential lists (for example, those that describe step-by-step procedures) appear as numbered lists.

Related Documents

The following documents provide more information about ZAATS.

- CLAS Configuration Location Analytics Software Server and Software Installation Guide
- ZAATS Tag Data and Numbering Guide

For the latest version of this guide and all guides, go to: www.zebra.com/support.

Service Information

If you have a problem with your equipment, contact Zebra Global Customer Support for your region. Contact information is available at: www.zebra.com/support.

When contacting support, please have the following information available:

- Serial number of the unit
- Model number or product name
- Software type and version number.

Zebra responds to calls by email, telephone or fax within the time limits set forth in support agreements.

If your problem cannot be solved by Zebra Customer Support, you may need to return your equipment for servicing and will be given specific directions. Zebra is not responsible for any damages incurred during shipment if the approved shipping container is not used. Shipping the units improperly can possibly void the warranty.

If you purchased your Zebra business product from a Zebra business partner, contact that business partner for support.

Provide Documentation Feedback

If you have comments, questions, or suggestions about this guide, send an email to EVM-Techdocs@zebra.com.

ZAATS Introduction

Introduction

Zebra's Advanced Asset Tracking System (ZAATS) provides continuous identification, location, and tracking of items tagged with passive UHF RFID tags conforming to the EPC™ Radio Frequency Identity Protocols Generation-2 UHF RFID Specification for RFID Air Interface standard. ZAATS is designed to enhance the efficiency and workflows of Zebra's customers' operations, which are increasingly focused on cohesive real-time data.

ZAATS consists of two primary components: the Real-Time Location System (RTLS) Services software, which contains the configuration, management, and location analytics components; and the ATR7000 overhead array readers.

Product Overview

Description and Features

ZAATS is a passive UHF RFID based asset tracking solution developed primarily for indoor warehousing, manufacturing, and logistics applications. It is based on the ATR7000 overhead RFID reader containing Zebra's proprietary advanced array architecture with integral antenna capable of steering beams and estimating the bearing to RFID tagged items with unprecedented accuracy and speed.

A summary of the key system and product features of ZAATS includes:

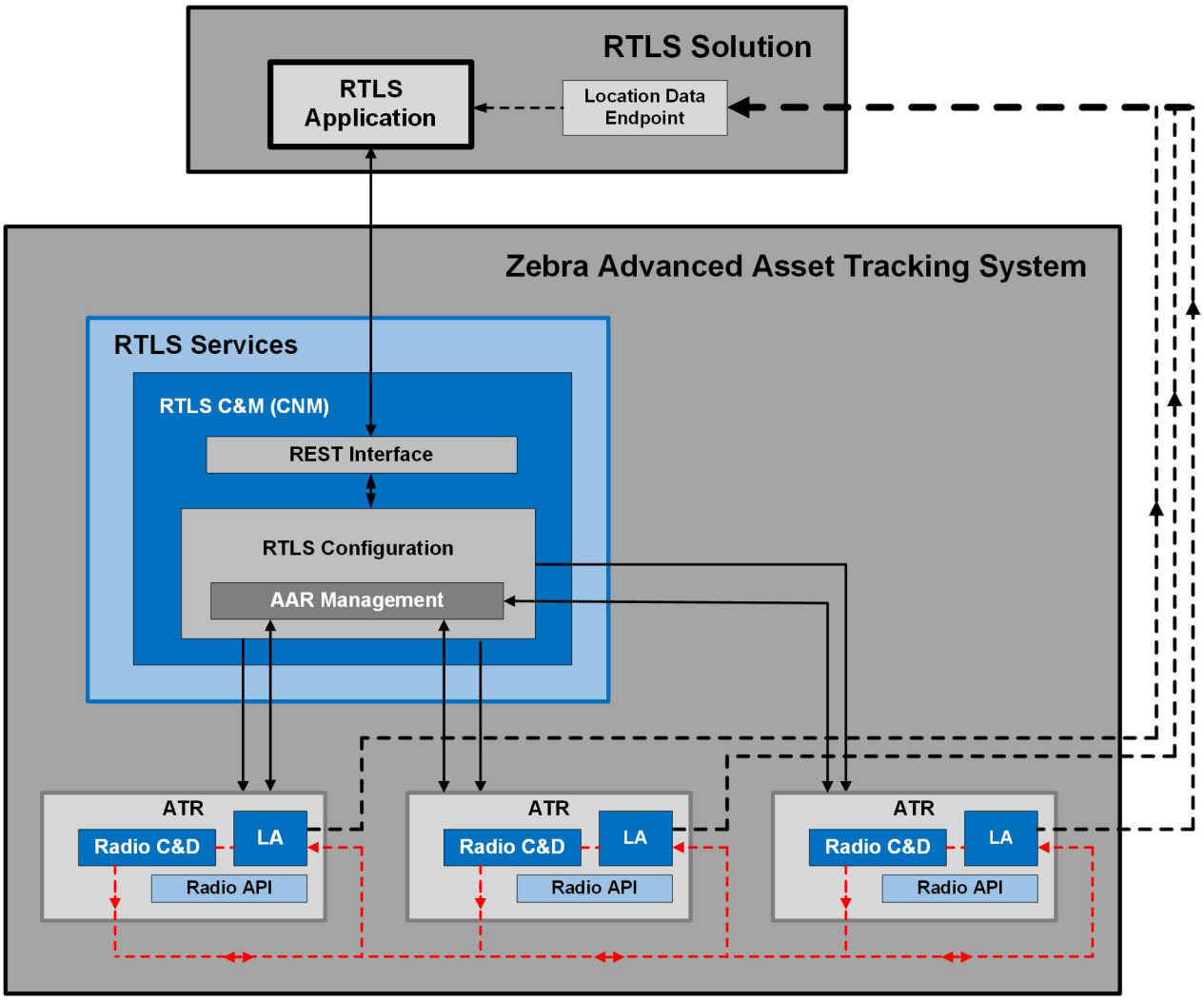
- A passive UHF RFID RTLS system that provides real-time identification and location data of tagged items for continuous asset monitoring.
- Configuration and Location Analytics Software (CLAS) that configures, controls, and manages the system, as well as a high-performance location analytics engine capable of providing up to 100,000 (1000 readers at 100 tps) tag location estimates per second with 2 ft typical accuracy.
- APIs to configure, manage, and control the ZAATS system using an HTTP-based RESTful interface.
- Docker container virtualization to simplify integration and deployment into end-user and partner applications.
- Software tools and documentation to facilitate system installation, including site planning, calibration, initial start-up, and deployment validation testing.

NOTE: Configuration and Location Analytics Software (CLAS) is needed to run RTLS Software. Configuration and Location Analytics Software (CLAS) is synonymous with RTLS Services software. The two terms are used interchangeably throughout this document.

Figure 1 illustrates the high level architecture of the ZAATS system showing the two main components of the ZAATS system:

- AARs (ATR7000 Advanced Readers)
- RTLS Services software.

Figure 1 Zebra Advanced Asset Tracking System (ZAATS)



In Figure 1, the solid lines correspond to configuration, control, and management interfaces. The dashed lines correspond to data interfaces. The dashed red lines carry tag ID bearing information that require high bandwidth and low-latency connections. Therefore, they typically reside on the same segment of a local area network.

ATR7000 Advanced Array Reader (AAR)

The ATR7000 Advanced Array Readers are EPC Gen2 readers with an integral phased array antenna capable of steering beams and estimating the bearing (angle of arrival) to EPC Gen2 tags. This product in the RFID portfolio is based on a Zebra proprietary advanced array architecture that provides unprecedented location accuracy and real time tracking of RFID tags.

A summary of the ATR7000 key product features are:

- Integral 14 element antenna array.
- Advanced multi-channel radio architecture provides accurate bearing estimations in a single read.
- Host platform compatible with Zebra's family of fixed RFID readers with support for embedded and external applications.
- Integration of Zebra's proprietary ASIC-based RFID radio.
- GPIO with external power for driving actuators and sensors.
- Support for several standard mounting options to simplify installation.
- Two power options: 802.3at Power over Ethernet (PoE+) or external +24 VDC power supply.
- Environmental specifications suitable for industrial and warehouse applications (-20°C to +55°C operation and IP51 sealing).

RTLS Services

RTLS Services (CLAS) serves as a data aggregator that executes location analytics to estimate the tag location and reports out unique tag ID, location, and time-stamp in real-time.

RTLS Services performs the following primary functions:

- Discovers readers on a local network.
- Configures each reader to read tags and report the estimated bearings.
- Estimates the tag location based on the bearings reported by the reader.
- Reports the location estimates to a location endpoint.
- Provides software interfaces to middleware applications that enable end-user solutions to associate items with identity, location and movement information, and delivers business logic to streamline operations or workflows.
- Provides an interface to end-users to configure and manage the RTLS system.
- Provides interface to manage and configure the ATR7000 readers in the facility.
- Performs license management functions for RTLS and CLAS software.

The RTLS Services software consists of three major components:

- RTLS Configuration and Management Server (CNM)
- Location Analytics (LA)
- Radio Control & Data (CND).

RTLS Services is deployed as a group of Docker containers. A container is the mechanism that minimizes operating system and hardware dependencies, and isolates RTLS from the other software components that comprise a solution, allowing them to coexist on the same infrastructure. It is expected that RTLS Services typically reside on the same physical server as the solution.

A description of these and other components important to system operation are below.

RTLS Configuration and Management (CNM)

As shown in Figure 1, RTLS Configuration and Management (CNM) is the primary component within RTLS Services responsible for managing and configuring the system, including system start and reset, reader discovery, initial and ongoing configuration of LA and CND, firmware and software upgrades, etc. CNM is the component of RTLS Services that is resident on the server.

RTLS also provides the configuration and management interfaces (API) to the solution software through a RESTful interface, a common framework found in enterprise environments.

Location Analytics (LA)

Figure 1 illustrates that Location Analytics (LA) is the primary component within RTLS Services responsible for aggregating bearing information received from the ATR7000 overhead readers, estimating x-y-z tag location, determining if a tag is moving (dynamic) or not moving (static), applying additional advanced algorithms that enhance static and dynamic location (tracking) accuracy, and reporting a final tag location estimate with metadata (EPC, timestamp, etc.) to a location data endpoint. LA also has the capability of combining raw bearing and location estimates from multiple RFID tags affixed to the same asset (for example, forklifts) to improve overall location accuracy and/or provide orientation and directionality information. The figure illustrates three AARs for simplicity, although, operation is designed to scale up to the maximum of 1000 readers per site. While LA is considered a component of RTLS Services, it is deployed by CNM to the readers at system start.

The interface between LA and the CND is optimized to be a high bandwidth, low latency one-way interface that carries only tag ID and bearing information, as indicated by the red arrows in the Figure 1.

Radio Control & Data (CND)

The Radio Control & Data (CND) component is a reader-based application (process) that configures, controls, and maintains a connection to the RFID radio (engine), receives tag bearing reports from the radio and passes them to LA, and ensures the timestamps on the bearing reports are synchronized to the system time source.

While CND is considered a component of RTLS Services, it is deployed by CNM to the readers at system start.

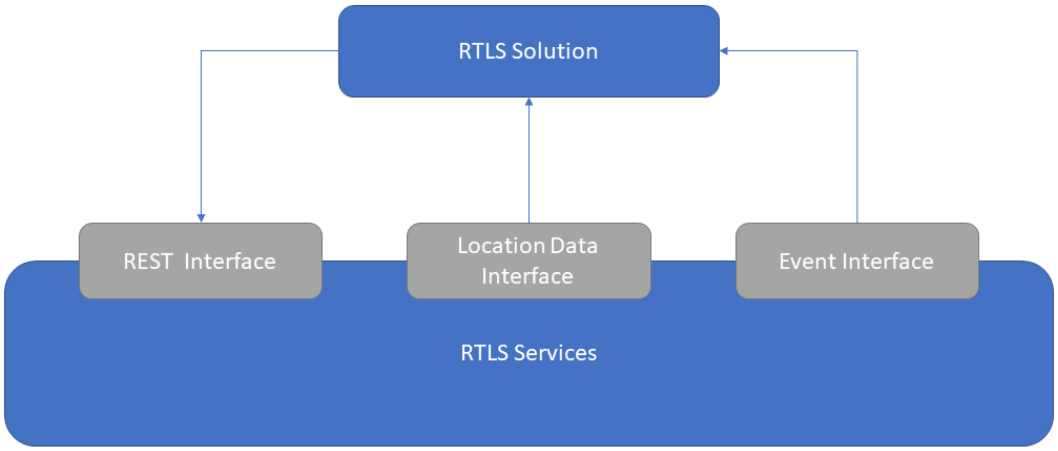
ZAATS Interfaces

ZAATS presents three main interfaces:

1. A REST based management interface.
2. A messaging stream interface for location data.
3. A messaging stream interface for health and monitoring events.

The ZAATS REST API allows applications to view, configure, manage, and monitor various system components in the RTLS system. The ZAATS location data interface allows the client application to consume the location data output by the ZAATS system. The ZAATS event interface allows the client application to consume the health and monitoring events data output by the ZAATS system.

Figure 2 Interface Overview



REST Interface

The REST Interface is the primary mechanism to configure and manage the ZAATS system. It supports the ability to query the version, status, configuration of the RTLS system; start and stop the system; and reboot the ATR7000 readers. It also supports setting user-defined filters specifying the frequency and format of reported tag data.

Location Data Interface

Location update messages are sent from the LA components within RTLS through the Location Data Interface to a Location Data Endpoint (MQTT server or a Kafka broker). The RTLS customer's middleware application can consume these location update messages from the Location Data Endpoint to transform information about asset location into solutions that enhance efficiency and workflows of end user operations.

Events Interface

Health and Configuration events notification messages are sent from CNM within RTLS through the Events Interface to a Event Endpoint (MQTT server or a Kafka broker). The RTLS customer's middleware application can consume these event notification messages from the Event Endpoint to implement solutions for monitoring of the RTLS system and raise alerts to end users about system events.

Configuring and Managing a ZAATS System

Introduction

Configuration of ZAATS and the CLAS software is performed using one of the following two methods:

- **Static configuration:** This method uses the rtls.conf file for configuring RTLS. Any changes made to this file takes effect only upon restart of the RTLS services software. For more information on the different configuration options available in the rtls.conf file, refer to the Configuring RTLS section in the CLAS Server and Software Installation Guide.
- **Dynamic configuration:** This method uses the REST API to configure the system. The configuration changes made using this method take effect immediately.

Most configuration of the RTLS system is done using the static method and the rtls.conf file as described in the CLAS Server and Software Installation Guide. Refer to Related Documents on page 6.

This chapter focuses on using the dynamic method to alter the configurable parameters of CLAS and describes how they are used to configure the system for specific operations.

Radio CND Configuration

The CND configuration can be changed with a POST call to /v2/rtls/config API and payload set to {“radio_c_and_d_config”: “<setting>”}.

The different values available for radio C&D configuration are as follows:

“sim”: radio_c_and_d_config should be set to sim to allow RTLS to operate with a simulator.

“bearing”: This will cause the CND to configure the radio to use the 49 beams in Figure 3 to read tags in the vicinity of the reader. By default, LHCP polarization is used for all the beams with 36 dbm EIRP.

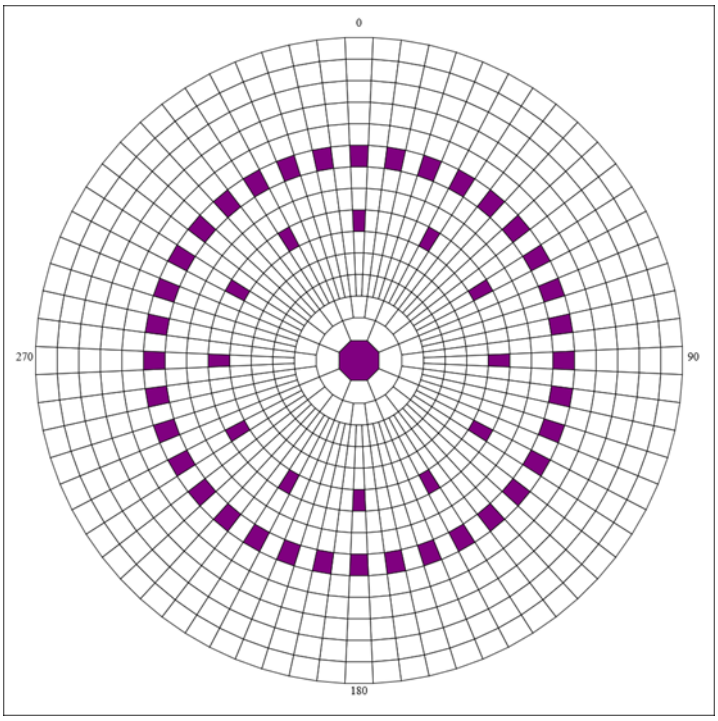
“bearings tx_pol=<value> tx_pwr=<value>”: this option allows the user to adjust some of the values used to configure the radio. It allows for configuration of the following parameters:

- **tx_pol:** This option can be used to select the primary beam polarization. The following values can be assigned to tx_pol:
  - theta
  - phi
  - theta_phi
  - lhcp
- **tx_pwr:** this option can be used to set the EIRP transmit power of primary beams in dbm.

Example: To set the primary and secondary beam polarization to phi and the transmit power to 33 dbm set the radio_c_and_d_config value in the JSON payload to:

{“radio_c_and_d_config”: “bearings tx_pol=phi tx_pwr=33”}

Figure 3 Azimuth-Elevation Chart Illustrating the 49 Primary Scan Beams Used by RTLS



In the figure above, azimuth is shown from 0º to 360º along the 12 o’clock position (due North),90º along the 3 o’clock position (due East), etc. Similarly, elevation is shown as 0 º at the center of the figure (boresight) out to 70 º at the outer edge (60 º is the maximum possible elevation scanning angle of the ATR7000). By default the beams are scanned using Left Hand Circular Polarization (LHCP) in a pseudo-randomized sequence covering 45 º elevation in 10 º steps, 30 º elevation in 30 º steps, plus boresight, for a total of 49 beams.

Location Analytics (LA) Configuration

LA allows configuration of the following settings:

- Filters
- Reporting
- Units
- Parameters

Configuring LA Filters

With every bearing update for each tag from an ATR7000, a new location estimate is calculated and can be sent to the Location Data Endpoint. However, in most cases, this is unnecessary as it produces redundant information. The list below includes different filters that can be enabled to filter the location estimate reports.

ID Filter: Only report if the tag (or multi-tagged item) matches the IDs described in the filter.

Distance Filter: Only report if the tag (or multi-tagged item) has moved more than x feet (or meters) since the previous reported location estimate report.

Time Filter: Only report if the tag (or multi-tagged item) has not been reported in the previous x seconds.

Velocity Filter: Only report if the tag (or multi-tagged item) is moving faster than x feet (or meters)/second.

Movement (Static or Dynamic) Filter: Only report if the tag (or multi-tagged item) is static or dynamic (moving) or ignore this aspect, and send a location update as per other filters.

Confidence Filter: Only report if the confidence of the tag's (or multi-tagged item's) location is greater than a certain percentage, x.

NOTE: The distance, time, velocity, and movement filters are "or"ed together. In other words, if any of the filters pass, a location estimate report is generated. If only a single filter is desired, the values for the other filters should be set to values so that they will not be triggered.

All of these filters can be set using the REST interface using the /v2/rtls/location_analytics/filters URI.

Configuring LA Units

The LA can be configured to work in feet or meters. This can be set in the rtls.conf file by setting "location_analytics_config_units" or by using the /v2/rtls/location_analytics/{laID}/units URI.

Configuring LA Fixed-Z Parameter

LA allows configuration of the Fixed Z parameter. When the LA gets a bearing report from a single ATR7000, it can produce a location estimate if it knows the height of the tag. If the height is known a priori (for example in a multi-tagged item), that height will be applied to determining the tag location estimate. If no information of the tag height is known, the LA assumes a height. This height can be explicitly set by using the /v2/rtls/location_analytics/{laID}/params URI. The units should match the units the LA is using.

Configuring LA Reporting

The tag location updates sent by the LA to the Kafka broker, includes the following fields by default.

- **epc_id**: hex string containing the tag's EPC ID
- **timestamp**: timestamp of the location update
- **position**: JSON object containing:
  - **x:** floating point number indicating the x position of the tag (or multi-tagged item) relative to the origin set by the RTLS system in whatever units (feet/meters) the RTLS system is using
  - **y:** floating point number indicating the y position of the tag (or multi-tagged item) relative to the origin set by the RTLS system in whatever units (feet/meters) the RTLS system is using
  - **z:** floating point number indicating the z position of the tag (or multi-tagged item) relative to the origin set by the RTLS system in whatever units (feet/meters) the RTLS system is using
- **confidence**: floating point number between 0 and 100 indicating the % confidence that the tag (or multi-tagged item) is at the location reported

In addition to the above default fields, LA can be configured to send the following fields as part of the location update.

- **report_source**: integer indicating the LA source of the location estimate report
- **static_or_dynamic**: string indicating whether the LA has determined the tag (or multi-tagged item) to be static or dynamic (possible values for the string are "static" or "dynamic")

- **velocity**: floating point number indicating the speed of the tag (or multi-tagged item)
- **direction**: floating point number indicating the direction of the multitag item
- **readers**: array of integers indicating the ATR7000 readers that read the tag
- **message_id**: integer indicating the message number

The exact set of fields that must be sent as part of the location update can be configured by using the /v2/rtls/location_analytics/{laID}/reporting API. For details, see the CLAS Server and Software Installation Guide.

The focus at the beginning of this chapter is on configuring the system using the dynamic method using the RESTful interface. The following table indicates the equivalent parameter settings in the rtls.conf file using the static method.

Table 1 Dynamic and Static Parameter Settings

	Dynamic		Static
	URI	**Body**	**rtls.conf**
CND			
	/v2/rtls/config	{“radio_c_and_d_config”: “<setting>”}	radio_c_and_d_config = <setting>
LA			
	/v2/rtls/location_analytics/		
	../filters	{“la_id_filter”: <filter>, “la_id_filter_mask”: <filter_mask>, “la_id_filter_num_bytes”: <num_bytes>	la_id_filter = <filter> la_id_filter_mask = <filter_mask> la_id_filter_num_bytes = <num_bytes>
	../filters	{“la_distance_filter”: <distance>}	la_time_filter = <distance>
	../filters	{“la_time_filter”: <duration>}	la_time_filter = <duration>
	../filters	{“la_velocity_filter”: <velocity>}	la_velocity_filter = <velocity>
	../filters	{“la_static_or_dynamic_filter”: <value>}	la_static_or_dynamic_filter= <value>
	../filters	{“la_confidence_filter”: <confidence>}	la_static_or_dynamic_filter= < confidence >
	../units	{“location_analytics_config_units”: <units>}	location_analytics_config_units=<units >
	../params	{“la_fixed_z”: <height>}	la_fixed_z=<height>
	../reporting	{<field_name>: <on_or_off>}	location_analytics_reporting_fields =<comma separated list of fields to enable>

RTLS Rest API

Introduction

This section describes the REST URI and HTTP methods used to configure and manage ZAATS.

The ZAATS REST API allows applications to view, configure, manage, and monitor various system components in the RTLS system.

RTLS REST API Features

RTLS REST API encompasses the following set of features and functionalities:

- Configuring the RTLS System
- Monitoring and managing the RTLS System
- Getting information about various RTLS components and tag locations
- Managing licenses
- Updating RTLS components.

The exact set of URIs and HTTP methods and the sequence in which they must be used to accomplish a specific task is detailed in the API Programming Examples chapter.

Viewing Live API Documentation

The documentation of the REST API response format is available as live documentation once the RTLS Services software is deployed and running. To view the API documentation, go to the following URL:

https://<rtls_server_ip>:<port>/doc

The variables in the URL should be substituted with appropriate values. The port variable must also be replaced with the port specified in .env during RTLS deployment.

NOTE: The default deployment of RTLS enables https. If RTLS Services software is deployed with https disabled (SSL_ENABLE option set to NO), then https must be changed to http.

For more information about deployment of RTLS Services software, see the CLAS Server and Software Installation Guide.

Accessing RTLS REST API

The RTLS REST API can be accessed from the following URI:

https://<rtls_server_ip>:<port>/<api_uri>

The API index below is a complete listing of the URIs and HTTP methods for the RTLS REST API.

API Index

The following tables list the APIs for all of the components of RTLS.

NOTE: To perform a group operation (on all components of a single type), the APIs must be called without an ID in the URI. However, to perform an operation on a single component (of a single type), the APIs must be called with an ID in the URI. For example, to access the status of all the ATR7000s in a single REST call, one must GET from URI: /v2/rtls/aar/status. However, to access the status of ATR7000 id #3, one must GET from URI: /v2/rtls/aar/3/status.

RTLS C&M REST API

Table 2 RTLS C&M REST API (base URI: /v2/rtls/)

Method	URI	Description
GET	/v2/rtls/version	Gets the version information of all RTLS components
GET	/v2/rtls/status	Gets the status information of all RTLS components
GET	/v2/rtls/config	Gets the current RTLS configuration
POST	/v2/rtls/config	Updates the RTLS configuration
GET	/v2/rtls/license	Gets the status of license of the RTLS system
DELETE	/v2/rtls/license	Return the license acquired by the RTLS system
GET	/v2/rtls/error/{errorNum}	Gets CNM error information
GET	/v2/rtls/warning/{warningNum}	Gets CNM warning information
GET	/v2/rtls/tag_location/{tagID}	Gets the location of a particular tag. The tagID parameter in this URL is mandatory to get the location estimates. An error is returned if the tagID parameter is not supplied.
GET	/v2/rtls/reference_tag/config/tags	Gets the list of all reference tags added to the system
POST	/v2/rtls/reference_tag/config/tags	Adds a reference tag to the RTLS system
DELETE	/v2/rtls/reference_tag/config/tags/{tagID}	Deletes the reference tag with ID equal to tagID from the RTLS system. The tagID parameter in this URL is mandatory to delete the reference tag. An error is returned if the tagID parameter is not supplied.
GET	/v2/rtls/reference_tag/config/error_threshold	Gets the error threshold value for reference tag monitoring in RTLS system

Method	URI	Description
POST	/v2/rtls/reference_tag/config/error_threshold	Sets the error threshold value for reference tag monitoring in RTLS system
GET	/v2/rtls/reference_tag/config/window	Gets the time window for reference tag monitoring
POST	/v2/rtls/reference_tag/config/window	Sets the time window for reference tag monitoring
GET	/v2/rtls/reference_tag/status	Gets the current reference tag statistics
POST	/v2/rtls/reader	Adds a new reader to the RTLS system
DELETE	/v2/rtls/reader	Deletes an existing reader from the RTLS system
POST	/v2/rtls/start	Starts all RTLS components
POST	/v2/rtls/stop	Stops all RTLS components
PUT	/v2/rtls/binary_image/{binaryFileName}	Uploads binary file to CNM

ATR7000 Management REST API

Table 3 ATR7000 Management REST API (base URI: /v2/rtls/aar/{aarID}/)

Method	URI	Description
GET	/v2/rtls/aar/{aarID}/version	Gets the ATR7000 version information
GET	/v2/rtls/aar/{aarID}/status	Gets the ATR7000 status information
GET	/v2/rtls/aar/{aarID}/location	Gets the ATR7000 location information
GET	/v2/rtls/aar/{aarID}/network	Gets the ATR7000 network information
POST	/v2/rtls/aar/{aarID}/reboot	Reboots the ATR7000
POST	/v2/rtls/aar/{aarID}/idle	Puts the ATR7000 in power idle mode
POST	/v2/rtls/aar/{aarID}/host_sw_update	Updates the ATR7000 host SW
GET	/v2/rtls/aar/{aarID}/host_sw_update_progress	Gets the progress of an ongoing ATR7000 host SW Update

Location Analytics REST API

Table 4 Location Analytics REST API (base URI: /v2/rtls/location_analytics/{laID}/)

Method	URI	Description
GET	/v2/rtls/location_analytics/{laID}/version	Gets the LA version information
GET	/v2/rtls/location_analytics/{laID}/status	Gets the LA status information
GET	/v2/rtls/location_analytics/{laID}/filters	Gets the LA filter information
POST	/v2/rtls/location_analytics/{laID}/filters	Sets the LA filters
GET	/v2/rtls/location_analytics/{laID}/reporting	Gets the LA reporting fields
POST	/v2/rtls/location_analytics/{laID}/reporting	Sets the LA reporting fields
GET	/v2/rtls/location_analytics/{laID}/multitag	Gets the LA multitag information
POST	/v2/rtls/location_analytics/{laID}/multitag	Sets the LA multitag information
GET	/v2/rtls/location_analytics/{laID}/tags	Gets the list of unique tags in the LA
GET	/v2/rtls/location_analytics/{laID}/units	Gets the units used by the LA
POST	/v2/rtls/location_analytics/{laID}/units	Sets the units used by the LA
GET	/v2/rtls/location_analytics/{laID}/params	Gets specific LA parameters
POST	/v2/rtls/location_analytics/{laID}/params	Sets specific LA parameters
GET	/v2/rtls/location_analytics/{laID}/logs	Gets a list of URLs to access the LA logs
POST	/v2/rtls/location_analytics/{laID}/logs	Enables/Disables the LA logs
GET	/v2/rtls/location_analytics/{laID}/error/{errorNum}	Gets LA error information
GET	/v2/rtls/location_analytics/{laID}/warning/{warningNum}	Gets LA warning information
POST	/v2/rtls/location_analytics/{laID}/update	Updates the LA subcomponent

CND Related REST API

Table 5 CND Related REST API (base URI: /v2/rtls/radio_c_and_d/{cndID}/)

Method	URI	Description
GET	/v2/rtls/radio_c_and_d/{cndID}/version	Gets the CND version information
GET	/v2/rtls/radio_c_and_d/{cndID}/status	Gets the CND status information
GET	/v2/rtls/radio_c_and_d/{cndID}/tags	Gets the list of unique tags in the CND
GET	/v2/rtls/radio_c_and_d/{cndID}/echo	Gets CND “echo” information
GET	/v2/rtls/radio_c_and_d/{cndID}/logs	Gets a list of URLs to access the CND logs
POST	/v2/rtls/radio_c_and_d/{cndID}/logs	Enables/Disables the CND logs
GET	/v2/rtls/radio_c_and_d/{cndID}/error/{errorNum}	Gets CND error information
GET	/v2/rtls/radio_c_and_d/{cndID}/warning/{warningNum}	Gets CND warning information
POST	/v2/rtls/radio_c_and_d/{cndID}/update	Updates the CND subcomponent
GET	/v2/rtls/radio_c_and_d/accelerometer	Gets the latest accelerometer readings of the reader
GET	/v2rtls/radio_c_and_d/foreign_tag	Gets the foreign tag added in RTLS
POST	/v2/rtls/radio_c_and_d/foreign_tag	Adds a foreign tag to RTLS
DELETE	/v2/rtls/radio_c_and_d/foreign_tag	Removes the foreign tag added to the system

API Programming Examples

Introduction

This section provides explanations of code samples provided in the samples folder of the RTLS release package. Most computer programming languages have libraries that support interfacing with REST APIs, however, Zebra has chosen Python to illustrate the use of REST API. All code samples are written in Python3 and implement a HTTP client that exercises various RTLS features. It also provides the formats for the location data that is published over the Kafka interface.

Depending on the task that needs to be performed, the examples in this section can be classified into the following categories:

- Using the Python Requests HTTP Client to Call RTLS APIs
- Getting Information about various RTLS Components and Tag Locations
- Configuring the RTLS System
- Monitoring and Managing the RTLS System
- Updating RTLS Components

Using the Python Requests HTTP Client to Call RTLS APIs

The first step is to have helper functions call RTLS REST APIs using different methods such as GET, POST, PUT, and DELETE.

The util.py file in the samples directory implements the HTTP client. It defines a class httpClient that contains three member functions named get, post, and put. The initializer of this class also sets the values of a few variables that are used by the member functions to make the requests.

Table 6 util.py Sample

```
self.http_scheme = "https"
self.rtls_server_ip = "10.17.129.70"
self.rtls_server_port = "80"
self.rtls_username = "rtlsadmin"
self.rtls_password = "Z@@t$R1l$"
self.verify_certificates = False
```

Change the values of the above variables to reflect the deployment parameters of RTLS. The verify_certificates variable should be set to false if proper CA certificates were not provided during RTLS deployment. Also, the username and password for RTLS should be changed if it is not deployed with default values.

RTLS REST APIs return one of three status codes for all HTTP requests as described below. Member functions in the client are intended to handle the returned status code appropriately.

1. **200: Operation successful.** In the case of GET, the response body contais the information requested. In the case of POST APIs, the response body contains either a JSON string or the body is empty depending on the API. Refer to the API documentation for more details.
2. **206: Partial response.** This status code is returned when a request is received to perform an operation on all readers, but the operation was successful on only a subset of all readers. In the case of GET APIs, the response body will contain the response from those readers on which the operation was successful. In the case of POST APIs, the response body consists of a list of those readers where the operation failed. Please refer to the API documentation for more details and JSON string formats in response body.
3. **500: Internal server error.** This status code is returned when the API request could not be performed due to an internal error. During this scenario, the response body almost always contains a JSON string with some information about the error encountered. This information is helpful to diagnose and debug the problem.

The error handling code for all member functions is as follows:

Table 7 Error Handling Code Sample

```
 if reply.status_code is requests.codes.ok:
    return reply.text
elif reply.status_code is requests.codes.partial:
    raise httpExceptions.PartialResponseError(reply.text)
else:
    raise Exception("RTLS returned http code:" + str(reply.status_code))
```

In this sample, the status code 206 for partial response is handled by defining a custom exception called PartialResponseError in the httpExceptions class. When http status code 500 is returned, a generic exception is raised. These exceptions must be handled in the calling functions to handle to the errors appropriately. Later examples demonstrate this.

A custom exception for a partial response is defined that can be handled by the calling functions appropriately.

Table 8 Exception for Partial Response Code Sample

```
class httpExceptions:
    class PartialResponseError(Exception):
        pass
```

Getting Information About Various RTLS Components

This section provides an overview of RTLS operations using GET APIs to get information and status about system as a whole or on a per component basis. Broadly the GET operations can be classified into two categories as follows:

- **Simple GET operations:** These operations perform a HTTP GET request on a URI to obtains info about the system. Examples include getting version information, health status, etc.

- **Compound GET operations:** These operations perform a HTTP GET request on a collection of URIs and combine the results to obtain further info. Examples include getting the locations of individual tags, getting the latest alert info if any in the system.

Simple GET Operations

Performing a GET operation on a URI will return the information requested in the JSON format. The samples folder in the RTLS packages consists several sample scripts that pull various information from RTLS by sending a GET request on an appropriate URI.

All sample scripts start by importing the httpClient and the httpExceptions classes from the util module defined earlier.

Table 9 Importing httpClient and httpExceptions class Code Sample

```
from util import httpClient
from util import httpExceptions
```

The httpClient class provides methods to perform a HTTP request to a given URI. It supports the GET, POST, PUT, and DELETE methods. To call these functions, the httpClient class must be instantiated as show below in Table 10.

Table 10 Create httpClient Object Code Sample

```
client = httpClient()
```

The instantiated httpClient object can be used to make HTTP requests to any RTLS REST API using the appropriate methods. To perform a GET request, the get() method is called with the URI as a parameter. If the GET operation is successful (status code is 200), then the get() method returns a serialized JSON response which can by loaded into a JSON object using the json module. In case of a partial response from RTLS (status code is 206), the get() method throws a PartialResponseError Exception.

The code snippet in Table 11 illustrates the GET operation on the "/v2/rtls/version" API to fetch the versions of the RTLS system and all its components. This code sample is present in the get_versions.py script in samples folder. Similarly, various information such as RTLS configuration, status, license status, aar status, etc. are fetched by passing the appropriate URI to the get() method.

Table 11 Get Version Information from the RTLS API Code Sample

```
try:
   ret = client.get("/v2/rtls/version")
   response = json.loads(ret)
   .....
except httpExceptions.PartialResponseError as response:
   .....
except Exception as e:
   .....
```

Compound GET Operations

There are many operations in RTLS that require the user to make multiple API calls to get more detailed information or combine information obtained from different calls. This section describes a few examples.

Getting ATR7000 Information

The example provided in the file get_aar_status.py demonstrates the sequence of calls necessary to get information about all the ATR7000 readers that are added to the RTLS system. The information obtained contains the host name of each ATR7000 and its location, MAC address, and Network IP. The user may choose to add more information about the readers such as the reader software versions to this example.

NOTE: Separate calls to RTLS are required to get location and network info of ATR7000s.

The function get_aar_info gets the network and location information of all the ATR7000s, combines them in a common list, and prints them. This starts by creating a httpClient object and then making two calls to /v2/rtls/aar/locations and /v2/rtls/aar/network APIs.

Table 12 Getting ATR7000 Infomation Code Sample

```
client = httpClient()
.....
try:
   aar_locations = json.loads(client.get("/v2/rtls/aar/location"))
   aar_addresses = json.loads(client.get("/v2/rtls/aar/network"))
```

The combination of the two lists is done based on the ATR7000 ID that is returned by both the APIs. We take the response from one API and iterate over all the elements. For each element in the first list, we look for an element in second list and match them by their ID. The merged info of an ATR7000 is then appended to a third list.

Table 13 Getting Merged Information from ATR7000 Code Sample

```
for location in aar_locations["locations"]:
   for address in aar_addresses["networks"]:
      if location["id"] == address["id"]:
         x = location.copy()
         x.update(address)
         aars.append(x)
```

Getting All Tag Locations

RTLS provides APIs to get the list of tags that are being read by the system. The location for each of these tags must be obtained separately. The code in get_tag_locations.py depicts how to get a list of all tags and then get the location estimate for each tag in the list.

First, start by importing the httpClient class to make the REST API calls.

Table 14 Import httpClient Code Sample

```
from util import httpClient
```

The function get_tags call /v2/rtls/location_analyitcs/tags API to get the list of tags for all location analytics blocks.

Table 15 Get List of Tags Code Sample

```
def get_tags():
   tag_list = []
   try:
      all_tags = json.loads(util.get("/v2/rtls/location_analytics/tags"))
```

It then prepares a single list that contains all the tags in the system, and returns the final list of tags.

Table 16 List of All Tags Code Sample

```
for tags_per_la in all_tags["tags"]:
   tag_list = tag_list + tags_per_la["tagID"]
```

The function get_tag_location gets the location of a single tag from RTLS. Then, it accepts a tag ID as a parameter and prints the location of each tag by sending a request to /v2/rtls/tag_location API. This API requires that the ID of the tag whose location is being queried be appended to the url.

For example, if the location of the tag with ID aaaabbbbcccc1111 is being queried then to get the tag locations, a get request to the URL /v2/rtls/tag_location/aaaabbbbcccc1111 must be made. This is shown in the code snippet below:

Table 17 Get Tag Location Code Sample

```
def get_tag_location(tag):
   try:
      loc = util.get("/v2/rtls/tag_location/" + tag)
      .....

   except Exception as e:
   .....
```

Obtain the list of tags in the system using the get_tags function and for each tag in the list call get_tag_location and pass the tag ID.

Table 18 Get Tags Function

```
tags = get_tags()
for tag in tags:
   loc = get_tag_location(tag)
   .....
```

Configuring RTLS System

This section explores code samples that configure various parameters in an RTLS system. The RTLS configuration broadly consists of the following:

- Changing a generic RTLS configuration such as the location/event endpoint type/address/topic
- Setting and resetting various LA filters

- Editing the location update report
- Adding Multi-tagged items
- Adding foreign tag
- Adding and configuring reference tags
- Adding and deleting readers

Configuring Location and Event Endpoints

To change any configuration parameters in RTLS, a POST HTTP request must be sent with appropriate payload when applicable. The payload accepted by all POST APIs is a JSON string with appropriate key value pairs. This is shown in this section with an example of configuring the location and events endpoints.

RTLS provides streaming interfaces for location and event data. Both location data and event data streams support Kafka and MQTT interfaces. Both the streams can be configured independently to either Kafka or MQTT. Also, the topic name on either of endpoint types can be configured. The code snippet in configure_endpoints.py shows how to configure the location and event endpoints using the RTLS configuration API. As shown in Table 19, the payload consists of a JSON string with values for location_endpoint_type, location_endpoint_addr, location_endpoint_topic and corresponding event endpoint parameters.

Table 19 Set Kafka Broker Address Sample Code

```
try:
   client.post("/v2/rtls/config", payload=json.dumps({"location_endpoint_addr":
   le_addr, "location_endpoint_type": le_type, "location_endpoint_topic": le_topic,
   "events_endpoint_type": ee_type, "events_endpoint_addr": ee_addr,
   "events_topic": ee_topic}))
   .....
except Exception as e:
   .....
```

Multi-tag Item Configuration

RTLS allows REST APIs to list all the multi-tagged items added to the system as well as add more multi-tagged items to the system. For more information on tag encoding for multi-tag item configuration, refer to the ZAATS Tag Data & Numbering Guide. The code snippet in list_n_add_multitag_items.py provides an example to list and add multi-tagged items to RTLS system for tracking. First, the multi-tag items that need to be added to the system must be defined. This is done by defining two multi-tag items mt_item_1 and mt_item_2 as shown in code snippet in Table 20.

Table 20 Define Multitag Items

```
mt_item_1 = {"general_manager_number": "abcdef0",
             "make_and_model_id": "12345",
             "make_and_model_name": "Toyota 8FGCUxx",
             "tag_locations": [{"id": 1,
                                "positions": {"x": -1.75,
                                              "y": 3.25,
                                              "z": 6.735}
                               },
                               {"id": 2,
                                "positions": {"x": 1.75,
                                              "y": 3.25,
                                              "z": 6.735}
                               },
                               {"id": 3,
                                "positions": {"x": -1.75,
                                              "y": -3.25,
                                              "z": 6.735}
                               }]
            }
```

To add the multi-tag item to RTLS, a POST request to /v2/rtls/location_analytics/multitag is made as shown in the previous examples. This API accepts only one multi-tag item at a time and the payload is a JSON string representing the multi-tag item.

NOTE: The /v2/rtls/location_analytics/multitag API only accepts one item at a time and has to be used accordingly.

Configuring Reference Tags

Adding reference tags to RTLS and configuring them properly helps keep track of location accuracy during its operation. In order to start monitoring RTLS location accuracy, users must add a few reference tags and their location to the RTLS system. When RTLS is started, the system continuously keeps track of the locations of the reference tags as computed by the system, and generates alerts if they deviate from their known location by more than the set threshold.

RTLS computes the overall system location accuracy over a window of time. The window is set in the rtls.conf file using the reference_tag_window variable or via /v2/rtls/reference_tag/config/window API. The units for the window are in minutes. When the overall system accuracy exceeds a threshold, an event is sent on the events channel notifying the user about the drop in system location accuracy.

The error threshold for generating reference tag alerts can be set in one of two ways.

- Set the global error threshold value by changing the “reference_tag_error_threshold” variable in the rtls.conf file or via the “/v2/rtls/reference_tag/config/error_threshold” URI. The global error threshold is used to send reference tag specific error threshold is not specified.

- Set the per tag error threshold at the time of adding the reference tag. The threshold set in this manner will override the global error threshold for reference tag alert generation for the specific tag. All other reference tags that do not have a tag specific error threshold will continue to use the global error threshold value.

The code snippet in configure_ref_tags.py shows how to configure the reference tag related parameters and add reference tags to the system.

The parameters for reference tags and the reference tags and their location are defined as shown in Table 21.

Table 21 Definition of reference tags and related parameters

```
global_error_thresh = 1
ref_tag_window = 2

ref_tags = [{
             "tag": "3175050952000030de000000",
             "x": 10,
             "y": 10,
             "z": 10,
             "error_threshold": 5
            },
            {
             "tag": "3175050952000030de000001",
             "x": 12,
             "y": 12,
             "z": 12,
             }]
```

As shown in Table 21 the script adds two reference tags to the system. One with a tag specific error threshold and another without it. It also sets the global error threshold to 1 (ft/meters as configured in rtls.conf file.), and the window to 2 minutes.

After setting the reference tag window and threshold, the reference tags are added one at a time by sending a POST request to "/v2/rtls/reference_tag/config/tags" API with a payload of JSON string that consists the reference tag defined earlier. After adding the reference tags, the sample script configure_ref_tags.py waits for the time period set for rolling window to elapse and then collects the reference tag statistics by sending a GET request to "/v2/rtls/reference_tag/status".

The script then waits for the rolling window to elapse before collecting the reference tag statistics.

Customize LA Reporting

RTLS allows customization of location reports sent to location endpoint by adding or removing fields from the location report. The REST API that can edit these fields is /v2/rtls/location_analytics/reporting. By default, when RTLS is started, the default fields are reported and fields can be added or removed later. The code snippet in add_remove_location_update_fields.py shows how to edit the fields in the location report.

First, a JSON object must be defined that specifies which fields must be turned on and which ones should be turned off. Then, the JSON object is passed as a payload to the /v2/rtls/location_analytics/reporting API.

Table 22 JSON Object Definition Code Sample

```
la_reporting = {"epc_id": "on",
                "timestamp": "on",
                "position": "on",
                "confidence": "on",
                "message_id": "on",
                "report_source": "on",
                "readers": "on",
                "direction": "on",
                "static_or_dynamic": "on",
                "velocity": "on"}
```

Set LA Filters

LA supports a list of filters that can be set or reset using the /v2/rtls/location_analytics/filters API. The code example in set_la_filters.py shows how to set the LA filters. For details on what values are allowed in each filter, please refer to the “live API documentation”.

NOTE: RTLS currently supports setting only one LA filter at a time. This is handled in the example by looping over a list of different filters and setting them individually.

Table 23 provides an example of a list of JSON defined objects. Each JSON object sets the filter to a certain value.

Table 23 JSON Object Lists Code Sample

```
la_filters = [{"time_thresh": 5},
              {"static_or_dynamic": "static_only"},
              {"velocity_thresh": 1000},
              {"id_filter": {"filter": "000000000000000000000000",
                             "mask": "000000000000000000000000",
                             "num_bytes": 12}},
               {"dist_thresh": 1},
             {"confidence_thresh": 50}]
```

Then, the set_la_filter function in the above list is iterated over and each individual filter is sent as a payload in POST request to the /v2/rtls/location_analyitcs/filters API.

Table 24 Payload Code Sample

```
for la_filter in la_filters:
      client.post("/v2/rtls/location_analytics/filters",
                  payload=json.dumps(la_filter))
```

To clear the filters, each individual filter (except static_or_dynamic filter) in the la_filters list should be set to 0.

To clear the static_or_dynamic filter the filter value should be set to **ignore**.

Monitoring and Managing RTLS System

Monitoring RTLS Health Status

The RTLS health API (/v2/rtls/health) provides a concise way of retrieving the health of all components in the RTLS system including the ATR7000 readers. The health API returns a health indicator string for each component in the system. The health indicator string can have three values: normal, warning, error. When a particular component health is in either warning or error conditions an appropriate API can be called to retrieve the details of alerts present in the component.

This is illustrated in the example script check_rtls_health.py. The check_rtls_health.py scripts retrieves the response from the health API to check for alert conditions on each component and make further API calls to get the alert details. The script starts off by sending a GET request to /v2/rtls/health API. In the returned health response object it iterates through each element to check if the component has an active alert. When a component is in an alert condition it checks the appropriate status API depending on the component in question to retrieve the alerts. For example, if there is an alert condition in the cnm component, then the script sends a GET request on the /v2/rtls/status API to get the alerts on the cnm component. If the health on either the radio, radio_host_connection or cnd is not normal then the script sends a GET request to the /v2/rtls/radio_c_and_d/{id}/status API to retrieve the alerts on the particular reader. Similarly, if there is an alert on the la component of the reader then a GET request to /v2/rtls/location_analytics/{id}/status API is made to fetch the alerts on the particular LA.

After checking all the components for alerts, the script finally clears the alerts by sending a POST request to the /v2/rtls/health API.

Table 25 provides a snapshot of the code snippet in the check_rtls_health.py script.

Table 25 Checking RTLS Health

```
def get_rtls_health():
    client = httpClient()
    try:
        ret = json.loads(client.get("/v2/rtls/health"))
        for component in ret:
            if component.startswith("cnm"):
                get_rtls_alerts()
            elif component.startswith("readers"):
                for reader in ret[component]:
                    for key in reader.keys():
                        if key.startswith("la"):
                            get_la_alerts(reader["id"])
                       elif key.startswith("cnd") or key.startswith("radio")
or key.startswith("host_radio-connection"):
                            get_cnd_alerts(reader["id"])
                        elif key.startswith("host"):
                            get_aar_alerts(reader["id"])
            elif component.startswith("central_la"):
                get_la_alerts(0)
    except httpExceptions.PartialResponseError as response:
        print("Recevied a partial response: " + str(response))
    except Exception as e:
        print("Error getting rtls health: " + str(e))
```

Get All RTLS Errors and Warnings

RTLS maintains a list of error and warning messages that occurred during the normal course of its operation. The number of errors and warnings in the system are returned as part of the status API in error_message_count and warning_message_count fields. RTLS maintains errors and warnings in a circular buffer, and the size of this buffer is set to hold 3000 errors and 3000 warnings. The code snippet in get_all_errors.py illustrates how to get a list of all RTLS errors and warnings.

Table 26 Get List of All RTLS Errors and Warnings Code Sample

```
try:
   return json.loads(client.get("/v2/rtls/error"))
except Exception as e:
   return None
```

Getting all warnings works the same as getting all errors. The code snippet in get_all_warnings.py replaces the URI with /v2/rtls/warning to get all the warnings.

Getting Latest RTLS Error and Warning Information

To get a specific error or warning, it must be fetched using its ID. As mentioned in the previous section, RTLS holds only 3000 errors/warnings. Since errors and warnings are stored as a zero-indexed list, the ID of the error or warning message lies somewhere in between 0 and 2999.

To get the latest error or warning information, it is necessary to know how many errors or warnings exist in the system using the /v2/rtls/status API. The status API returns the number of errors and warnings that are currently held by RTLS. Therefore, to get the latest error or warning, the /v2/rtls/error or /v2/rtls/warning API must be called along with the ID of the error or warning in question. The ID must be one less than the count for error or warning returned in response to the status API request. This is illustrated more clearly in get_last_error.py and get_last_warning.py files. (Table 27)

In both examples, a GET request is sent to v2/rtls/status API error_message_count or warning_message_count is extracted from the response. The example below displays this process for error messages.

Table 27 Get Latest RTLS Error and Warning Information Code Sample

```
status = json.loads(client.get("/v2/rtls/status"))
if status["error_message_count"] > 0:
   rtls_errors = client.get("/v2/rtls/error/" +
                    str(status["error_message_count"] - 1))
   rtls_latest_error = json.loads(rtls_errors)
```

Reboot All Readers

To reboot all readers, a POST API call has to be made to /v2/rtls/aar/reboot. This is a group AAR Management API and returns response codes, 200, 206, and 500. In cases of a partial success, (status code 206) the response body consists of a JSON object that provides a list of readers that failed to reboot. This is handled in the code by catching the PartialResponseError exception.

Table 28 Reboot All Readers Code Sample

```
try:
   client.post("/v2/rtls/aar/reboot")
except httpExceptions.PartialResponseError as partial:
   print("Following AARs failed to reboot: " +
      json.dumps(json.loads(str(partial))["failed"]))
except Exception as e:
   .....
```

Restart RTLS

To restart the RTLS system, a sequence of two API calls are required to stop the RTLS and then start the RTLS. Please note that restarting RTLS does not restart the RTLS containers, this merely restarts the RTLS components (CND and LA). Furthermore, issuing a stop will stop the LA application, whereas the CND application continues to run. The stop command merely causes the CND to stop reading tags and publishing tag bearings. The start operation starts the LA application, but since CND is already running, the start operation will instruct the CND to start reading tags and publish the bearing information. The code snippet in restart_rtls.py shows the RTLS restart operation.

Updating RTLS Components

To update any RTLS component, a sequence of operations need to be performed.

1. Upload the updated version of the component to RTLS
2. Stop RTLS
3. Trigger Update
4. Monitor Update and wait until update is finished
5. Start RTLS

The same process is followed in all updates as detailed in the Uploading Files to RTLS and Updating Reader Software sections.

Uploading Files to RTLS

As a first step in all update operations, the binary image of the component must be uploaded to RTLS server. To do so, one must use the PUT method on /v2/rtls/binary_image/ API. Some details to consider about this API are as follows:

1. The name of the file being uploaded should be appended to the URL at the time of calling it. For example, if the file being uploaded is aar_new_sw.zip, then the URL will be /v2/rtls/binary_image/aar_new_sw.zip. The same file name must be supplied in the request body at the time of triggering the update. This will be demonstrated in the code samples below.
2. The name of the files being uploaded can include letters, numbers, underscore and a dot for file name extensions.

The request body consists of the contents of the file as the file upload is handled as a multi-part encoded file. The below example shows how to upload a file. The same method is used in all update procedures that involve uploading a binary file.

Table 29 Updating Files to RTLS Code Sample

```
try:
    file = open("/home/zebra/new_binary_file.ext", 'rb')
    files = {'file': file}
    client.put("/v2/rtls/binary_image/new_binary_file.ext", files=files)
    return True
except Exception as e:
    .....
    return False
```

Updating Reader Software

The RTLS REST interface provides APIs to update various components. ATR7000 reader software is one of the things that can be updated using RTLS. To update the reader software, the update package must be uploaded to the RTLS server in ZIP format. To create an update package place all the files in a folder and zip the folder. The name of the ZIP file should be the name of the folder at the top level and a .zip extension. Before starting the update, the RTLS server should be stopped while an update command is issued to start the update. Progress of the update is tracked until all readers are updated. To track the progress the /v2/rtls/aar/host_sw_update_progress API is used.

The code sample for updating reader software is in update_aar_software.py.

1. Upload the binary image to RTLS server, as explained in the previous section.
2. Stop RTLS by issuing a HTTP POST request to /v2/rtls/stop URL.
3. Issue a POST request to the /v2/rtls/aar/host_sw_update URL to start the update.

The host_sw_update API must be given a JSON object in its payload that specifies the name of the file that was previously uploaded. For example, if the name of the zip file uploaded to RTLS is new_aar_image.zip, then a payload of JSON object {"name": "new_aar_image.zip"} must be sent along with the request. This is shown in the following code snippet.

Table 30 Update Reader Software Code Sample

```
try:
   client.post("/v2/rtls/aar/host_sw_update",
               payload=json.dumps({"name": "new_aar_image.zip"}))
except httpExceptions.PartialResponseError as response:
   .....
except Exception as e:
   .....
```

NOTE: This sample updates the reader software on all of the ATR7000 readers managed by RTLS. To update a single ATR7000 reader, the URI should include the AAR id, as described in the note shown in the API Index.

To get the progress of update, a GET HTTP request is sent to URL /v2/rtls/aar/host_sw_update_progress and extracts the progress on all readers from the response. The progress is obtained repeatedly until all readers are finished updating, hence the progress API is called within a while loop. When readers start updating, they report a progress that starts from 0 and increments to 100. They report -1 when the update is no longer in progress. The function wait_for_update breaks out of the while loop when all readers report a progress of -1 (not updating).

Table 31 Monitoring Host Software Update Progress Code Sample

```
try:
      .....
      while not update_finished:
      .....
      progresses = client.get("/v2/rtls/aar/host_sw_update_progress")
      progresses = json.loads(progresses)
      for progress in progresses["progresses"]:
         if int(progress["progress"]) > -1:
            aar_pending = True
            .....
      if not aar_pending:
         update_finished = True
except httpExceptions.PartialResponseError as response:
   .....
except Exception as e:
   .....
```

NOTE: Readers reboot after the update is completed. RTLS waits for the readers to reboot and reinitializes the readers after they are up.

After the readers finish updating, they reboot automatically to finish the update. The update continues even after the readers reboot. This process takes about 5 minutes to complete. The sample code waits for a duration of 5 minutes and then starts RTLS.

RTLS Location and Event Reporting

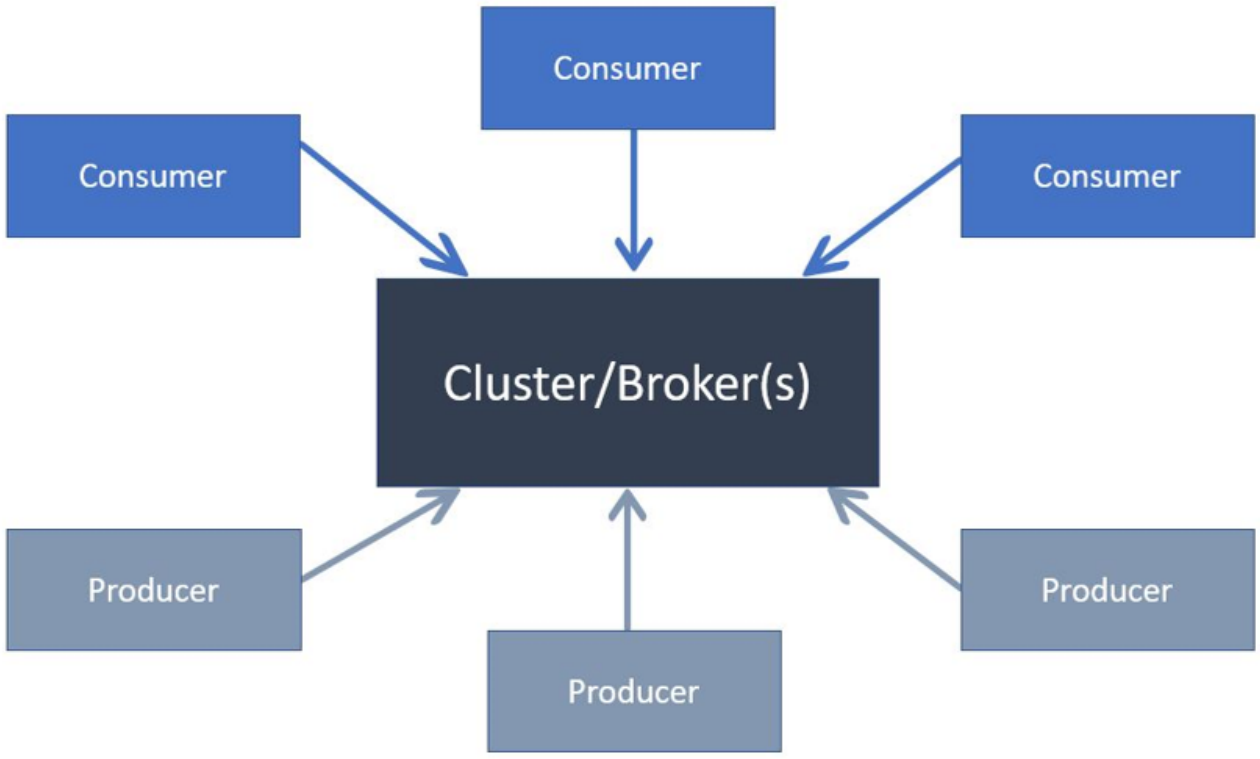
Messaging Platform Overview

A messaging platform typically consists of three main components:

- The producer produces messages that are sent to the broker.
- The broker accepts messages from producers, manages messages, and allows consumers to access messages.
- The consumer pulls messages from the broker.

RTLS supports two messaging platforms: MQTT and Kafka. Access details about MQTT at http://mqtt.org/, and access details about Kafka at https://kafka.apache.org/.

Figure 4 Messaging Platform Diagram



ZAATS supplies only the message producers. The RTLS solution supplies the broker(s) and consumer(s) to read the messages being pushed by RTLS into the message broker.

Location Reporting

As described in ZAATS Introduction, the Location Analytics (LA) block of the RTLS processes bearing reports of a tag from multiple ATR7000s and combines them to calculate a location estimate for that tag. The resulting tag location estimate is reported to a Location Endpoint.

A Location Endpoint in RTLS refers to a message broker to which the location estimates are published on the specified topic. RTLS supports Kafka and MQTT message brokers as Location Endpoints.

In RTLS, each LA contains a producer that produces tag location report messages that are streamed to a message broker at the IP address: TCP port configured in the rtls.conf file.

Message Formats

The format of each location estimate update is a JSON object encapsulated in a string. The format of the JSON object as follows:

```
{
    "message_id": integer,
    "epc_id": string containing hex digits,
    "timestamp": string containing ISO8601 compliant timestamp,
    "confidence": float,
    "position":{
        "x": float,
        "y": float,
        "z": float
    }
    "static_or_dynamic": string,
    "report_source": integer,
    "velocity": float,
    "direction": float,
    "readers": [integer,...]
}
```

Example of a Location Update

NOTE: The ordering of the fields in the JSON object is not guaranteed.

```
{
    "epc_id":"3175050952000190210000000",
    "timestamp":"2017-08-27T07:05:50.304+0000",
    "confidence": 70.00,
    "position":{
        "x":190.08,
        "y":57.58,
        "z":4.55
    }
}
```

Event Reporting

The RTLS software continuously monitors various components of the RTLS system and generates events in case of adverse events. Like the location reporting, RTLS event reporting also sends messages to an events endpoint to be consumed by the user. RTLS supports Kafka and MQTT as the events interface, and can be configured using rtls.conf file or via the REST API.

RTLS generates two types of events: Warning and Error. A Warning event indicates that the RTLS CNM has noticed by adverse behavior of one or more RTLS components, and is taking action to remedy it. An Error event is generated when RTLS CNM fails to remedy a Warning event or if an event that CNM cannot handle has occurred.

The following components are monitored by CNM and can be the source of events generated by CNM:

- CND: Any warning/error encountered by the CND is propagated via the Events interface.
- LA: Any warning/error encountered by the LA is propagated via the Events interface.
- AAR: Any adverse ATR7000 behavior is reported by CNM as Warning or Error
- CNM: Any events encountered in CNM itself is also sent by CNM to the Event interface.

Message Formats

The format of each event report is a JSON object encapsulated in a string. The format of the JSON object is as follows:

```
{
   "source-type": string,
   "source-id": integer,
   "event-type": string,
   "event-id": integer,
   "timestamp": string,
   "event-info": string
}
```

Field Description

The following table contains details of all the fields in the message.

Table 32 Field Descriptions

Field	Description	Values
source-type	This field is a string that indicates the RTLS component that is the source of the event.	“cnm”, “aar”, “cnd”, “la”
source-id	This field contains an integer that is the ID of the source mentioned in the source-type field. For Example: if the CND with ID 0 has an error then the source-type field will contain “cnd” and the source-id field will contain 0.	A valid integer
event-type	This field is a string that specifies the type of event that has occurred.	“warning”,”error”
event-id	The integer in this field is the ID of the event that has occurred. The IDs are maintained differently for different type of events and for different components.	A valid integer
timestamp	A string containing the timestamp of when the event was generated.	A valid timestamp string
event-info	A string containing the description of the event.	

Example of Event Reporting

```
{
   "source-type": "cnm",
   "source-id": 0,
   "event-type": "error",
   "event-id": 0,
   "timestamp":
}
```