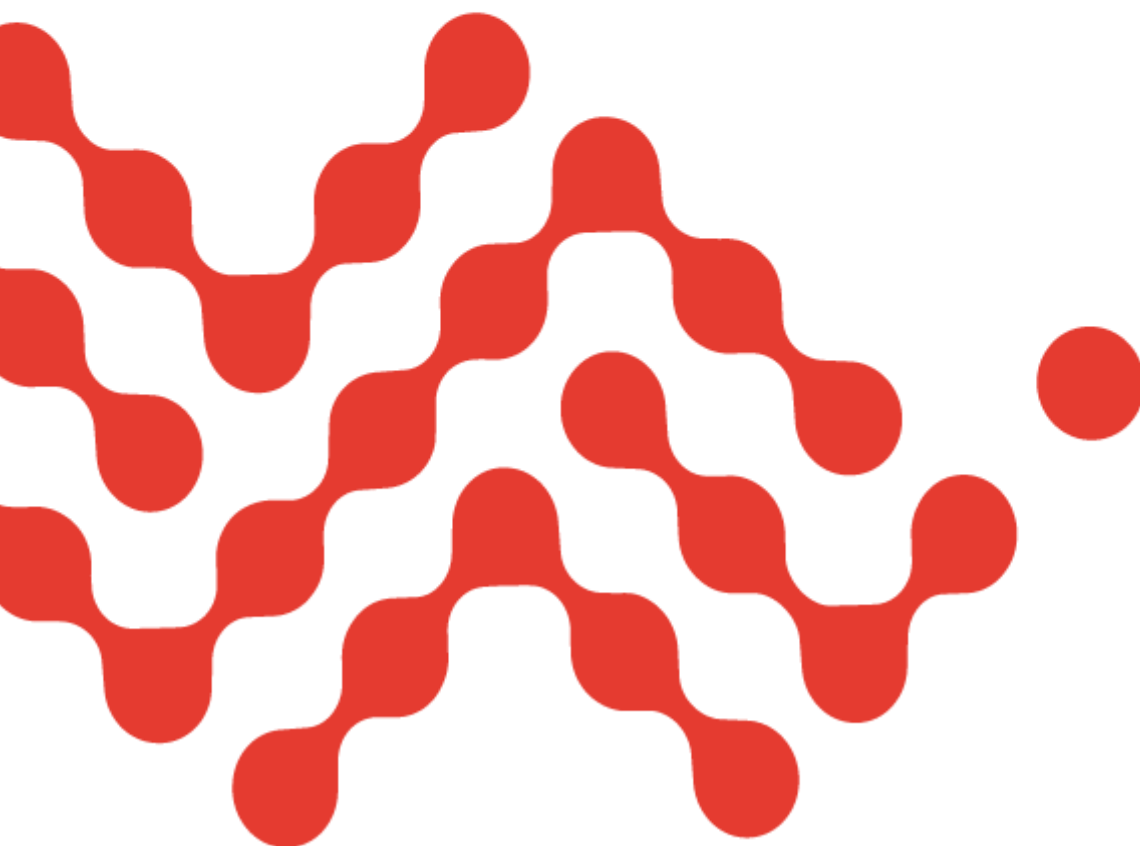


ReadyAgent

API User Guide





Copyright

© 2010 Sierra Wireless. All rights reserved.

Trademarks

AirCard® and Watcher® are registered trademarks of Sierra Wireless. Sierra Wireless™, AirPrime™, AirLink™, AirVantage™ and the Sierra Wireless logo are trademarks of Sierra Wireless.

wavecom®, , , inSIM®, WAVECOM®, WISMO®, Wireless Microprocessor®, Wireless CPU®, Open AT® are filed or registered trademarks of Sierra Wireless S.A. in France and/or in other countries.

Other trademarks are the property of the respective owners.

Document history

Date	Version	Auteur	Description
11/11/08	Draft	D. FRANCOIS	Document creation
04/09/09	Draft	D. FRANCOIS	Full update
21/04/09	1.0	C. BUGOT	Remove language examples (they are factorized into code samples) Add more general information on the libraries
30/04/09	1.0	C. BUGOT	Add information on software update and OSGi bundle
31/03/10	2.0	C. BUGOT	Changes the template of the document Add new features from the Agent libraries
30/06/10	2.1	C. BUGOT	Minor edits for ReadyAgent R2.0 Release

Content table

ReadyAgent	1
API User Guide	1
Document history	2
1. Introduction	5
2. Reference document	5
3. Platform overview	5
3.1. General architecture	5
3.2. Embedded architecture	6
3.3. Embedded agent	7
3.4. User libraries	8
4. Protocol library	8
4.1. Overview	8
4.2. Usage	8
5. Communication library	9
5.1. Overview	9
5.2. Usage	9
5.2.1. Initialization and registering	9
5.2.2. Sending and receiving data	9
5.2.3. Software update	9
5.2.4. Sending and receiving SMS	10
5.2.5. Setting and getting ReadyAgent internal variables	10
6. AWTDA High Level library	10
6.1. Overview	10
6.2. Usage	11
6.2.1. Initialization	11
6.2.2. Data containers	11
6.2.3. Sending events	13
6.2.4. Receiving data and commands	13
6.2.5. Accessing lower level APIs	13
7. OSGi	14

8. Code samples	14
8.1. Description	14
8.2. Java	15
8.3. C Asynchronous.....	15
8.4. C Synchronous.....	15
8.5. Lua.....	15

1. Introduction

This document explains the usage of the Platform Embedded API. The API is available in several programming languages and programming pattern: Java, C Asynchronous, C synchronous, and Lua. The aim is to be generic, but whenever it is necessary, language specific notes can still be present in this document.

In order to be more concrete, a code sample for every programming language/pattern is provided along with this document.

2. Reference document

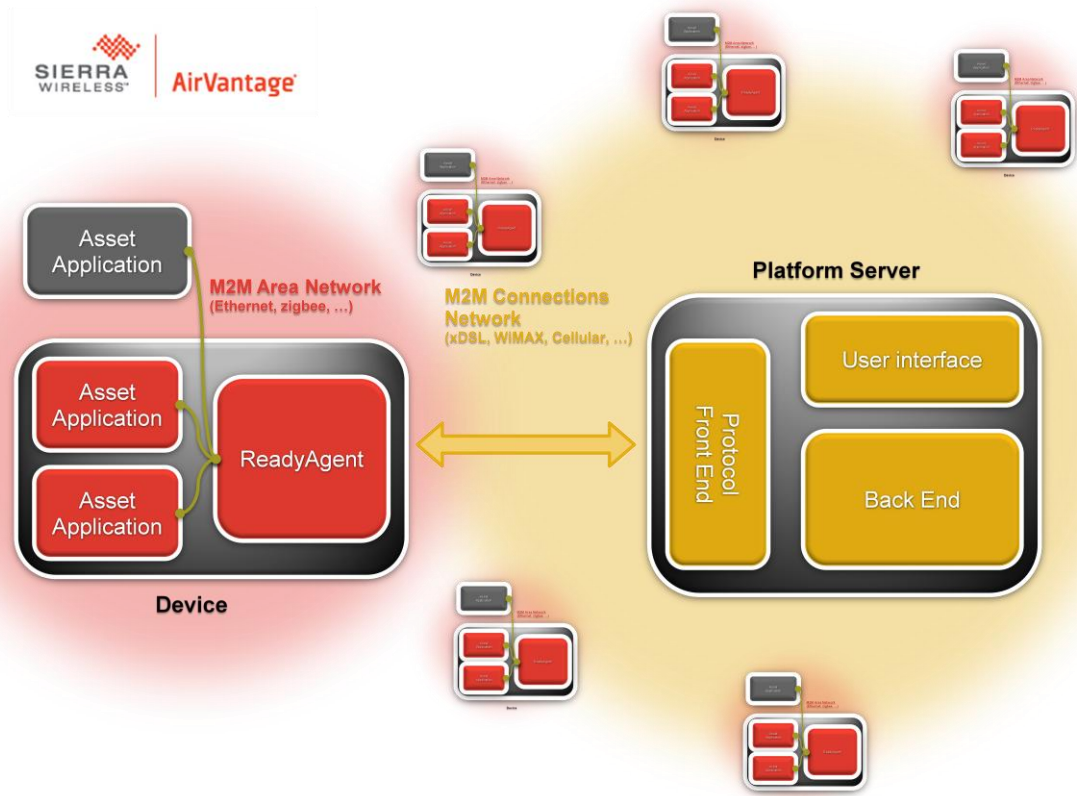
- [REF-1] PLT08 SP01 AWTDA-Protocol
- [REF-2] Java API: javadoc
- [REF-3] C API: doxygen
- [REF-4] Lua API (Lua text files)
- [REF-5] Monitoring Engine
- [REF-6] ReadyAgent Configuration

3. Platform overview

3.1. General architecture

The platform enables applicative data management, device management and an easy way to configure a set of devices according to the costumers needs.

The platform can be divided into two parts, the platform server and the device side. Here is a simple representation of those two sides.



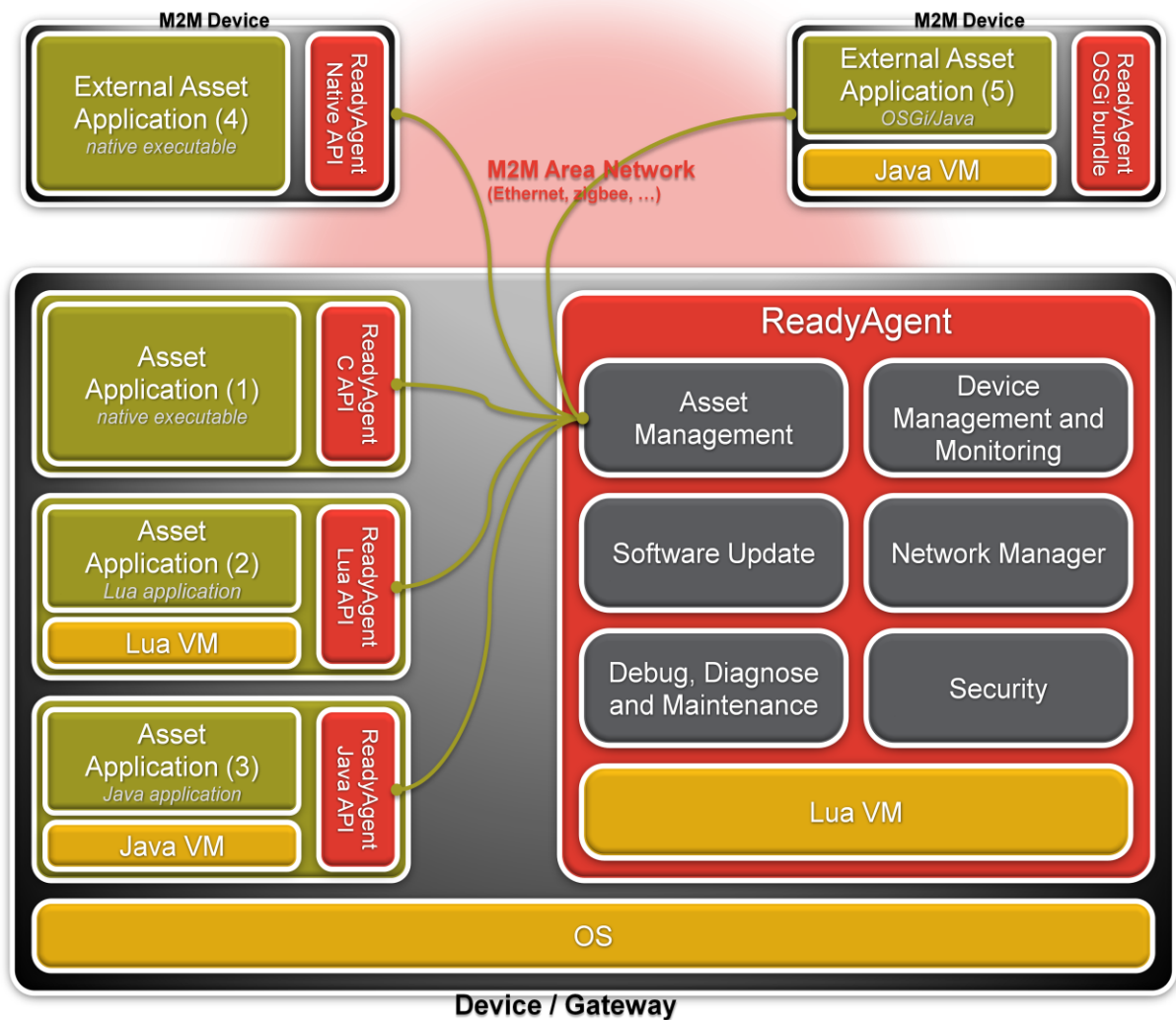
The embedded application encodes the data and events and sends it to the embedded ReadyAgent through an IPC (a socket for example). The agent adds some message envelope (compression, authentication ...), and sends it to the platform server through HTTP or another transport protocol.

The platform server front-end receives and decodes it, and sends it to the application data store where data and events will be stored.

3.2. Embedded architecture

The next figure illustrates the typical embedded architecture on a Linux target. This architecture is composed of the ReadyAgent, ReadyAgent libraries and one or more applications.

Depending on the embedded hardware capabilities, OSGi framework or Java runtime environment may not be available. Also the ReadyAgent may run in a different process (as a daemon) or be statically linked in a monolithic software.



3.3. Embedded agent

The embedded ReadyAgent is in charge of dispatching incoming messages to applications and to send user applicative data and events to the platform server. The exchange between applications and the ReadyAgent is done through an IPC. This guaranties a high separation between application code and system code.

Before sending a message to the server, the agent adds a message envelope (compression, authentication, headers...). All data received by the Agent have to be serialized according to the protocol specification described in document [REF-1]. This serialization can be done with the AWTDA Protocol Library and the High-Level Library described below.

3.4. User libraries

As written above, three libraries are provided with the ReadyAgent.

The protocol library is used to encode /decode data according to the protocol specification see [REF-1]. This library is available in the language used for the client application.

The communication library is an abstraction of the IPC used to communicate with the agent. It provides a consistent API in all languages. Moreover this library provides system level functionalities as device configuration and sending/receiving serialized messages from the platform. This library is generally not used directly by the user, but rather used by the AWTDA High Level API.

Finally the AWTDA High-Level library provides a simple way of sending data, events to the server; and receiving data and commands from the server. This library handles all the complexity of the AWTDA protocol structure. The API provided by this library is the one the user usually wants to use.

4. Protocol library

4.1. Overview

The protocol library (AwtDaProtocol) provides AWTDA Objects definition and all needed functions to create, serialize and de-serialize them. See [REF-1] for more details about the protocol specification.

This library is currently implemented in Java, C and Lua. The C version of the library provides a simple object oriented API and simplified garbage collection system: it uses internal reference counters in order to know if an object needs to be freed. See the library API specification for the language you are interested in.

4.2. Usage

You have to use this library when you want to receive and send data using the AWTDA protocol.

For sending data, the process is first to define and instantiate all objects, then to serialize the objects. This binary stream can be used for exchanging with an AWTDA aware peer.

The process is exactly the symmetric when you receive data. First you have to de-serialize the stream, and then you can parse the objects and retrieve the content of the message.

This library is context less, no initialization is needed. However it is not thread safe at object level (meaning you cannot use the same object at the same time in different threads), but

that is very unlikely. If it is ever needed to share the same object between threads, the user has to ensure that the threads do not access the object at the same time.

In the general case this library is not used directly for serialization itself (as the AWTDA High Level library takes care of it). See the code samples for examples.

5. Communication library

5.1. Overview

The communication library (AwtCom) provides a transparent access to the agent, and thus the platform server. This library is usually linked with the client application (when the operating system allows creating processes) and provides native language API in order to communicate over the IPC (a socket on most systems) with the agent.

This library is protocol independent, and can be used for AWTDA or OMADM related operations. Moreover this library provides system specific features as sending SMS or monitoring hardware values.

5.2. Usage

5.2.1. Initialization and registering

When initializing the library context, it will automatically register to the agent with the given asset identifier (*assetid*). The asset identifier is the root element of the message path. An asset does not necessarily represent a real piece of hardware; it is merely a representation of the global application architecture.

5.2.2. Sending and receiving data

Once you have initialized an AwtCom context, you can send and receive data. For the moment only AWTDA protocol data is supported, even though the API does not restrict the kind of data that is sent or received.

Those data are not necessarily sent to the platform server synchronously, this depends on the ReadyAgent server connection policy. However, if you want to force a connection to the server, an API is available for such action.

5.2.3. Software update

The AwtCom library provides easy software update mechanism that is well coupled with the Platform Server. The final update process (replace a binary file, update a text configuration file, upgrade a device firmware, etc.) is really dependent on the application, and thus it is let at user disclosure. There are exceptions though: for OSGi bundle management, the

ReadyAgent provides an end to end solution, including bundle management (see OSGi section below).

The Platform Server and ReadyAgent are actually in charge of the transfer and notification of new software update packages. The system guaranties that the software package transfer is safe: different options are available: checksums, authentication, and encryption.

In order to receive Software Update notifications, the user have to register a listener on the AwtCom context. The notification provides a **packageName**, a **packageVersion** and a **url** that specifies where the file is located (on the local file system).

5.2.4. Sending and receiving SMS

The library provides an API in order to register a listener for SMS reception. In order to register for SMS reception the user has to give a pattern. If the emitter of a received SMS matches the pattern, then this listener is called.

Moreover a simple send SMS API is provided in order to send 8bits encoded SMS.

The ReadyAgent relays incoming and outgoing SMS, handling the concatenation if the SMS content is larger than the single SMS capacity. There is no restriction on the emitter/recipient of an SMS at the ReadyAgent itself, though it must be connected to a GSM modem with an activated SIM card.

5.2.5. Setting and getting ReadyAgent internal variables

The ReadyAgent has a list of internal variables that can be manipulated by a user application. For instance those variables allow interacting with the Monitoring Engine, ReadyAgent configuration.

The provided getVariables and setVariables API allows accessing those ReadyAgent internal variables.

See Monitoring and ReadyAgent configuration documents for more information.

6. AWTDA High Level library

6.1. Overview

The AWTDA high level API (AwtDaHL) provides tools to easily create, serialize and send AWTDA Objects to the agent. The API also offers the possibility to receive all incoming messages selected by type and path. This library uses the communication and protocol libraries internally.

This API has been designed to ease the conception and development of custom applications, without having the burden of dealing with all protocol specific optimizations and connection states.

As the communication library, this library is linked with the client application so to provide a full abstraction of the agent and platform server.

6.2. Usage

6.2.1. Initialization

When initializing the library you will instantiate a library context. This object will be used to manipulate, send and receive data from the server.

The asset identifier given in the context initialization API will be used to prefix all sent data. Likewise, when registering message reception listeners, only messages addressed to that asset identifier will be received.

6.2.2. Data containers

In order to send data to the server, you will need to create data containers, fill in the containers, and then flush the containers to the agent.

There are two types of containers.

1.1.1.1. *Uncorrelated time stamped containers*

Those containers enable transport of uncorrelated lists of time stamped values. For instance you could transmit the *power* and the *temperature* variables, each one with different timestamps and different number of values.

power	
timestamp	values
1233786292	10052
1233787285	10821
1233789120	10845
1233790152	11002

temperature	
timestamp	values
1233786100	20
1233787201	19
1233789156	20
1233792568	17
1233792726	16
1233795810	18

In order to optimize the transport of the data, there are 5 pre-defined time stamped structures:

1. **TimestampedValue**: send one *value* with one *timestamp*.
2. **TimestampedValues**: send *n values*, one *startTime* timestamp and *n-1 timestamp deltas*.
3. **TimestampedDeltas**: send one numerical *startValue*, *n-1 value deltas*, one *startTime* timestamp and *n-1 timestamp deltas*.
4. **PeriodicValues**: send *n values*, one *startTime* timestamp and a *period*.
5. **PeriodicDeltas**: send one numerical *startValue*, *n-1 value deltas*, one *startTime* timestamp and a *period*.

1.1.1.2. Correlated containers

Those containers enable transport of structured correlated list of values. This means that all variables are synchronous and that the lists have the same number of values. Moreover if the data needs to be time stamped, a timestamp column can be added.

timestamp	power	temperature
1233786100	10052	20
1233787201	10821	19
1233789156	10845	20
1233792568	11002	17
1233792726	12025	16

The process to send data is the following:

1. create a Data Container;
2. fill in data into the container;
3. add the container into the instantiated context;
4. flush the data;

Steps from 1 to 3 can be repeated multiple times for different data containers. Please note that after step 3, the container is locked and no more data can be filled in until the flush is done. Also, a call to flush API causes the containers to be emptied.

Finally, the user can optionally request that the sent data get acknowledged by the peer (server). This is useful when critical server behavior is expected from the embedded application. To enable this feature, the user only has to provide a listener when adding the Data Container to the AwtDaHL instance.

6.2.3. Sending events

This library provides an API to send one or several events to the platform server. The events are composed of a *timestamp*, *code*, *type* and can have optional attached *data*. The *code*, *type* and *data* have application dependent values.

Sent events can be acknowledged in the same manner it is done for data sending.

6.2.4. Receiving data and commands

This library provides an API for registering listeners on Command and Data reception. The user specifies the path and message type he wants to be notified from, and the library will call the listener on message reception.

A Command message is formed of a command name (a string) and a list of arguments (number, string or binary).

A Data message has actually the same format as the ones that are sent to the server. However, only correlated data message types with a single row are allowed. Those data messages are usually used to write remote configuration parameters.

6.2.5. Accessing lower level APIs

For some application and use cases, it may be necessary to access lower level or protocol independent functionalities that are not exported into the AWTDA high level API (force a ReadyAgent<->Server connection, sending/receiving SMS, etc.).

For that purpose the high level API can return a handle on the AwtCom library that is used internally. The user can then use AwtCom API directly in order to perform specific actions.

7. OSGi

When using an OSGi framework, the AwtPlatform bundle is provided. This bundle exports all services of the three above libraries and is also in charge of handling update of all bundles in the OSGi framework.

In order to use a service of the ReadyAgent libraries, you have to import necessary packages from the AwtPlatform bundle.

The AwtPlatform bundle is in charge of updating applicative bundle or “core” bundles of the frameworks. The AwtPlatform bundle is registered as an AwtSoftwareUpdateListener in the Communication library (with the *assetid* “osgi”). In order to update any bundle a SoftwareUpdate event as to be sent from the platform server on the *assetid* “osgi” and with a *packageName* equals to the name of the bundle to update or install.

8. Code samples

8.1. Description

The code samples show how to use the AWTDA High Level library in different programming languages. The implemented application simulates an asset, called “house” with two rooms. Some data and events are sent for those two rooms.

The bedroom preset temperature of the heating is sent. This data is sent with an AwtTimeStampedValue and does not need any acknowledgment. An event “Temperature too hot” (code: 105) is then sent (time stamped one minute later). This event needs not to be acknowledged by the server.

A record of the five last living room temperature measurements (5 min period) is sent; it uses AwtPeriodicDeltas and does not need an acknowledgment. The event “Window opened” (code: 106) is sent. This event needs to be acknowledged by the server, a trace into a listener indicates the reception of this acknowledgement.

A Command listener waits for a Command Message reception on the asset “house”, a trace indicates the name of the received command.

A Data Writing listener waits for a Data Message reception on the path “house.bedroom”, a trace indicates the content of the data received.

Application Data Model:

Path	Data type	Name	Period	Values	Ack needed
house.bedroom	AwtTimeStampedValue	preset-temperature	One time	19	No
house.living-room	AwtPeriodicDeltas	temperature	5 minutes	16,17,18,17,19	No

Application Event Model:

Path	Name	Code	data	Ack needed
house.bedroom	Temperature too hot	105	« Temperature too hot »	No
house.living-room	Window opened	106	« Window opened »	Yes

The given code samples use as few as possible system calls in order to simplify the understanding of the AWTDA High Level API. Thus all actions happen synchronously without timers (apart from the listeners that are actually called on message reception). For the same reason, the timestamps used may be hard-coded values.

8.2. Java

The java application example is located into the directory CodeSamples/java.

8.3. C Asynchronous

The C Asynchronous application example is located into the directory CodeSamples/C/async.

8.4. C Synchronous

The C Synchronous application example is located into the directory CodeSamples/C/sync.

8.5. Lua

The Lua application example is located into the directory CodeSamples/lua