Developer's Guide

Ayla Embedded Agent for Embedded Systems

Version: 3.0

Date Released: April 19, 2018

Document Number: AY006DAR6-3

Ayla Networks

Copyright Statement

Trademarks Statement

Referenced Documents

Ayla Networks does not supply all documents that are referenced in this document with the equipment. Ayla Networks reserves the right to decide which documents are supplied with products and services.

Contact Information

Ayla Networks TECHNICAL SUPPORT and SALES

Contact Technical Support: https://support.aylanetworks.com
or via email at support@aylanetworks.com

Contact Sales: https://www.aylanetworks.com/company/contact-us

Ayla Networks REGIONAL OFFICES

GREATER CHINA
Shenzhen
Room 310-311
City University of Hong Kong Research Institute Building
No. 8 Yuexing 1st Road
High-Tech Industrial Park
Nanshan District
Shenzhen, China
Phone: 0755-86581520

HEADQUARTERS
Silicon Valley
4250 Burton Drive, Suite 100
Santa Clara, CA 95054
United States
Phone: +1 408 830 9844
Fax: +1 408 716 2621

JAPAN
Room #701
No. 2 Ueno Building 3-7-18
Shin-Yokohama, Kohoku Ward
Yokohama City, 222-0033 Japan
Phone: 045-594-8406

EUROPE
Munich
Ludwigstr. 8
D-80539 München
Germany

TAIWAN
Taipei
5F No. 250 Sec. 1
Neihu Road, Neihu District
Taipei 11493, Taiwan

For a complete contact list of our offices in the US, China, Europe, Taiwan, and Japan, click:
https://www.aylanetworks.com/company/contact-us

Table of Contents

1 Introduction

The Ayla Embedded Agent enables modules to connect to the Ayla Cloud Services using a white box development model. This embedded agent can be integrated with embedded networking and OS software to run on a Wi-Fi communications module. The customer application code runs on the module with the Ayla Embedded Agent and has access to all the features and services of the OS and networking stack on the module.

NOTE The Ayla Embedded Agent was previously called the Ayla Device Agent (ADA), and there are still references to ADA in our code and end-user documentation.

1.1 About this Document

This document describes the design and implementation of the Ayla Embedded Agent software for use in embedded microprocessor systems. The document includes specifications on features, algorithms, software interfaces, and integration considerations for the Ayla Embedded Agent.

NOTE This document does not describe the use of the Ayla Embedded Agent on any particular Software Development Kit (SDK). Refer to the developer guides on support.aylanetworks.com for the specific SDK for your Ayla Embedded Agent package.

1.2 Intended Audience

This document is for software developers who wish to design and implement an embedded system with connectivity to the Ayla Device Service (ADS) and mobile applications.

1.3 Related Documentation

Refer to the following documentation on support.aylanetworks.com as additional resources to this document:

- *Ayla Product Development Overview - Creating IoT Products: Overall Flow* (AY006GOV1) for an overview of the steps needed to bring any IoT product to market with Ayla Networks.
- *Ayla Cloud Service Overview* (AY006OCS0) for descriptions of the cloud service components and architecture.
- *Ayla Module and MCU Interface Specification (*AY006MSP0), which primarily addresses another method of using Ayla-enabled modules with a separate

application MCU, much of the document does not apply to the use of the Ayla Embedded Agent, but the document provides useful descriptions of the properties, the TLVs that make up schedules, and the client configuration.

- *Ayla Device Wi-Fi for Embedded Systems* (AY006DAW6), which describes the design and implementation of the Ayla Device Wi-Fi (ADW) software. Refer to the section on the Ayla Device Wi-Fi support library.
- *Ayla Command Line Interface (CLI) User Manual* (AY006UCL3)
- *Ayla Developer Portal User's Guide* (AY006UDP3-2)
- *Ayla OEM Dashboard User's Guide* (AY006UDB3-4)

1.4 Document Conventions

This document uses these Ayla documentation conventions:

- For APIs, functions returning `0` on success may return `-1` or some other non-zero value on failure. If the type returned is `enum ada_err`, a successful return is zero and the error returns is an error number less than zero.
- Text that you type (i.e. commands) and the contents of downloaded files are shown as:

```
cd wmsdk_bundle-3.1.16.1
tar xfz ada-wmsdk-src-1.0.tgz
```

- Function prototypes, function names, variables, structure names and members, and other code fragments are shown in `courier new`, a fixed-width font.
- Network paths, file paths, menu paths and the like are shown in `courier new` font and each point that you have to click to navigate to the next is separated by "`/`."
- Information on hazards that could damage a product or cause data loss is shown as:

Ensure that the appropriate data buffering is accounted for in deployed devices, where loss of data is critical to the core functionality or the services provided by the systems.

1.5 Revision History

Table 1 provides a summary of updates to the content of this document.

Table 1: Revision History of Content Updates and Changes

Revision #	Date (yyyy-mm-dd)	Change
1.0	2015-08-31	Initial version
1.1	2016-05-01	Added Wi-Fi interface requirements
2.0	2016-12-17	Updates for ADA 1.2
2.1	2018-04-09	Added file property transfer interfaces

1.6 Abbreviations and Acronyms

The following acronyms are used in this document.

ADA	Ayla Device Agent. This is a legacy term for Ayla Embedded Agent. The ADA acronym is still used in CLI commands and sources, and descriptions of either of these may refer to the Ayla Embedded Agent as the "device agent."
ADS	Ayla Device Service (the cloud service)
ADW	Ayla Device Wi-Fi (library for embedded systems)
ANS	Ayla Notification Service
CRC	Cyclic Redundancy Check
DNS	Domain Name Server
EVB	Evaluation Board
LwIP	Lightweight Internet Protocol
mDNS	multicast Domain Name Server
OTA	Over-the-Air firmware update
REST	Representative State Transfer, a method of web-based APIs
RTC	Real-time Clock
RTOS	Real-time operating system
SDK	Software Developer Kit

SPI	Serial Peripheral Interface
SSID	Service Set Identifier
TCP	Transmission Control Protocol
TLS	Transport Layer Security, also known as SSL
UART	Universal asynchronous receiver/transmitter
UDP	User Datagram Protocol

1.7 Glossary

Ayla Device Service	The Ayla Device Service (ADS) provides a way to deliver control and status information between any device and the web or a mobile application. This information is transferred in data items called properties. The communication may take place either over the Internet from the device to ADS and then to the mobile application, or directly from the device to the mobile application over the local-area network (LAN).
LAN Mode	Local-Area Network mode allows connection and control from a mobile device on the same IP subnet as the device's.
Ayla Production Agent (Black Box)	Ayla Production Agents are fully-managed, Ayla-enabled modules intended to be used as-is by the manufacturer. These agents are pre-loaded on Wi-Fi communication modules to enable the device to connect to the Ayla Cloud Service. An external processor running the customer application can connect to the module via SPI or UART. The module includes the Ayla Production Agent along with the networking and security software. Some primary characteristics and benefits are: • Available for embedded solutions. • Provides the fastest time to market for OEMs • No smart gateway required. • No custom gateway or other forms of communication agent software, including QA required regardless of the type of end-device. • No need to hire embedded engineers. • Any microcontroller-based system can be enabled with cloud connectivity.

2 Ayla Embedded Agent

Ayla Networks has four different device agents that run on IoT devices or gateways and incorporate a fully optimized networking stack, along with the additional protocols needed to connect devices to the Ayla Cloud. Developers using almost any microcontroller or operating system can use Ayla-enabled connectivity modules to enable cloud capabilities over any type of networking protocol. The four device agents are:

- Production Agent (Black Box). Refer to Section 1 of this document for a high-level description.
- Embedded Agent (White Box), which is described in this section.
- Linux Agent. Refer to *Linux Agent Setup* (AY006ULA6) and *Device Application for Linux* (AY006DAC6) on support.aylanetworks.com.
- Generic Gateway. Refer to *Generic Gateway Application for Linux* (AY006DAG6) on support.aylanetworks.com.

This section provides a description of the Ayla Embedded Agent (White Box) and its benefits and requirements.

2.1 Features and Benefits

The Ayla Embedded Agent is essentially embedded code integrated with embedded networking and OS software to run on a Wi-Fi communications module. This allows OEMs to create their own firmware solution for secure connectivity to Ayla Services. The customer application code runs on this module with the Ayla Embedded Agent, and has access to all of the features and services of the OS and networking stack on the module. Some primary characteristics are:

- Available for embedded or LINUX solutions.
- Targeted for specific Wi-Fi manufacturers and their supported SDKs.
- The Ayla Embedded Agent is available as a library or source.
- Secure, reliable connectivity to the Ayla Cloud Service.
- LAN mode support – direct mobile-to-device communications.
- Support for file property transfer (upload and download)
- Support for OTA image management and distribution.
- On-Device schedules for automated control.
- Wi-Fi setup for specific Wi-Fi modules, and example application code.
- Well-equipped for applications with existing RTOS and networking.
- APIs and data representation are similar to the Ayla Production Agent (Black Box).
- The Ayla White-Box-based Cloud Agent's modular design allows code for additional functions to be included as needed.

2.2 OEM Requirements

Though the Ayla Embedded Agent (White Box) is a type of endpoint that allows for a more complex and versatile device than the Black Box class of devices, the development effort is significantly longer for OEMs and therefore results in longer time to market than the Black-Box modules. OEMs are responsible for all device side applications including:

- Host application implementation.
- OS features, such as tasks, timers, memory management.
- Networking stack, including TLS.
- Security routines for authentication and encryption.
- Integration of the Ayla Embedded Agent into the OS and network stack.

3 Interfaces to the Ayla Platform

The Ayla Embedded Agent relies on the functionality provided by the platform and supplied by the product integrator. These include:

- C library functions
- RTOS functions
- TCP/IP network
- Connection monitor: indicate to ADA when the network is available and not.
- If Wi-Fi is used, the platform must provide for Wi-Fi setup and monitoring.
- TLS / SSL client
- HTTP client
- HTTP server
- A configuration system to persist configuration information over power cycles.
- Console output for debug logging
- A CLI or other method to configure the device during manufacture
- Firmware update methods (OTA), if desired
- Real-time clock

4 Major Components of the Embedded Agents

The Ayla Embedded Agent is divided into the subsystems shown in the diagram in Figure 1.

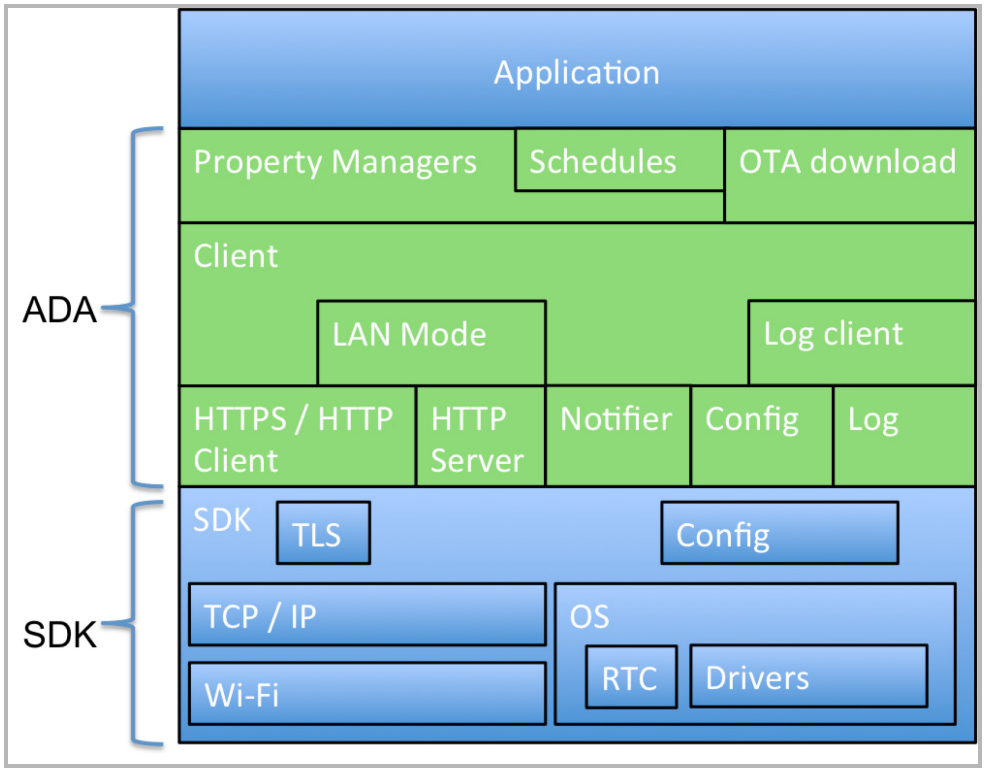

Figure 1: The Subsystems of the Embedded Agents

NOTE As mentioned in the Section 1, the Ayla Embedded Agent is often referred to as ADA, which stands for Ayla Device Agent.

4.1 ADA Client

The ADA client manages the connection to the Ayla Device Service (ADS). The ADA client receives property updates and commands and then sends property updates and responses to commands, using RESTful HTTP interfaces over TLS.

4.2 The Notify Subsystem

The notify subsystem in the Ayla Embedded Agent manages the protocol used by the Ayla Notification Service (ANS). This maintains a UDP flow with ANS so that the ANS may send events indicating when the ADS has property updates or commands for the client. This subsystem sends a periodic UDP packet, typically every 30 to 90 seconds to keep the firewall open for notifications from the ANS. This also informs the ANS and ADS of the on-line status of the device.

4.3 LAN Client

The LAN client operates as an HTTP service on the local network. The RESTful interface presents to mobile apps that allow them to read and write properties and send some commands to the device. The device and mobile apps use a shared secret during a key-exchange to create a session key used to encrypt subsequent traffic. The mobile client sends a keep-alive HTTP PUT command periodically.

4.4 HTTP Client

The HTTP client subsystem of the Ayla Embedded Agent provides a common interface to the HTTP client in the platform's SDK. The client and LAN client use it for their RESTful interfaces.

4.5 Property Subsystem

The property subsystem collects the interfaces between the application and the client to allow parts of the application to control the Ayla Device Service (ADS) properties. As the client receives properties, these properties are sent to various property managers and then the client sends them to the application. As the application receives the new property values, the application may call property manager interfaces to send the new values to the client.

4.6 Schedules

The schedule subsystem provides the ability for the device to change a property value on a specified time schedule. For example, a schedule sent to the device from the cloud could entail turning on lights from 7 PM to 11 PM every weeknight. As long as the time is known, schedules respond even when Internet connectivity is down.

On the ADS, schedules are distinct from properties, but in the Ayla Embedded Agent schedules are treated as just another type of property.

4.7 Configuration Subsystem

The configuration subsystem interfaces to the SDK's or the application's particular method for saving data in persistent storage between boots of the device. The configuration is organized as a set of name / value pairs. The names are hierarchically oriented and specified by ASCII tokens separated by slashes. The values can be binary data up to 1 Kilobyte or more, but are typically much shorter UTF-8 strings.

4.8 Logging

The Ayla Embedded Agent has a logging system used by each of the agent's subsystems. Log messages can be enabled or disabled by subsystem and severity. Normally, error, warning, and informational messages are enabled on all subsystems. In addition, debug or verbose-debug messages can be enabled.

4.9 CLI Interfaces

The Ayla Embedded Agent provides a small number of CLI interfaces that may or may not be used by the application. These interfaces allow some debugging and manufacturing setup of the device.

4.10 Over-the-Air Updates

The Ayla Embedded Agent and ADS provide the ability to download OTA updates to the device. The format of the OTA and method of applying the update is left up to the SDK and application implementer. The Ayla Embedded Agent has APIs, which notify the application that an OTA is available, let the agent start the download, receive the update sections, and report the status of the update back to the ADS.

5 Multithreading Considerations

The Ayla Embedded Agent can be used in a number of multithreading models. The simplest, from a multi-threading perspective, is the asynchronous model, where almost all activity is done in a single thread. The black-box model uses this with the TCP/IP thread from LwIP. This conserves memory by doing everything on one stack, but increases complexity by forcing all code to be non-blocking and event-driven.

The second model, which is used in all currently supported SDKs, runs the agent code in its own thread. Events and calls from other threads are queued to be performed the agent thread. The agent thread may block while doing HTTP client calls, and these should be protected by time limits. In this model, responses from HTTP client requests come back in the same thread.

6 Memory Requirements

The memory requirements for the Ayla Embedded Agent are feature-dependent, and depend on the SDK. This section estimates the memory required on ARM Cortex-M3 or -M4 processors, not including the size of the platform SDK, which must provide certain features, such as an TLS client. The TLS client can significantly add to the base memory required by the platform when used by other applications. These memory estimates also only approximate the amount of code needed in the application to provide the initial configuration support needed by the Ayla Embedded Agent.

For a system in which code cannot be directly executed from flash, the code requires space in both flash and in RAM. Code space includes any significant initialized-data space as well. Data space is almost entirely uninitialized data, so only requires RAM. The code space estimates can be used to determine the amount of flash needed for OTA updates.

NOTE Remember to include any additional space is needed for Wi-Fi firmware or Web server images.

Tables 2 and 3 provide estimates for the memory required by the Ayla Embedded Agent code and data. If LAN support, schedules, and OTA are not desired, you can deduct the memory required by those three subsystems. The sizes for OTA are only the memory required for downloading the image, not for saving or applying the update. The schedule support is almost entirely code. The data requirement depends on the number of schedules supported.

Table 2: The Ayla Embedded Agent Code

Subsystem	Code size (bytes)	Data size (bytes)
ADA cloud client	51000	7700
ADA LAN client	11000	4600
Ayla Schedules	5100	32
OTA support	2100	0
Heap	0	20000
Stacks	0	10000
SSL Certs (DER alternative)	3600	0
SSL Certs (PEM alternative)	5100	0
TOTAL	77900	42332

Table 3: Typical Platform

Subsystem	Code size (bytes)	Data size (bytes)
RTOS	51000	7700
SDK and drivers		4600
TCP/IP	5100	32
SSL client	2100	0
HTTP client	0	20000
HTTP server	0	10000
Heap (not including ADA)	-	0
Stacks (not including ADA)	-	-
Ayla Embedded Agent	-	-
Minimal demo application	-	-
TOTAL	58,200	42,332

NOTE The memory allocated for packets in TCP/IP (LwIP, in this example) is more than absolutely necessary. It may be acceptable to reduce it by 10k to 15k bytes.

7 Application Programming Interfaces (APIs)

This section describes the following (as they pertain to the Ayla Embedded Agent) and the associated APIs:

- Client
- Property Subsystem
- Alternative Property Subsystems
- Ayla Schedules
- Logging Subsystem
- Over-the-Air Updates
- CLI Interface

7.1 Client

The platform initialization calls `ada_init()` to initialize the Ayla Embedded Agent before waiting for a Wi-Fi or network connection to be established. Once a network connection is established, the `ada_client_up()` call is made to start the connection to the ADS. Following are the APIs used:

ada_init()

```
int ada_init(void)
```

This call initializes the Ayla Embedded Agent. The call returns `0` on success.

This API should be called after the platform (including Wi-Fi) is initialized, but before waiting for a Wi-Fi or network connection to be established. The `ada_conf` structure should be initialized first. `ada_init()` initializes the Ayla Embedded Agent software real-time clock (RTC) and sets the hardware RTC if it is set to a time before a minimum value required for the TLS certificates. This also sets the Ayla Embedded Agent logging options. If necessary, `ada_init()` creates the client thread. Finally, `ada_init()` sets up mDNS to advertise the device for the mobile application.

ada_client_up()

```
int ada_client_up()
```

The platform calls `ada_client_up()` when the local network is connected and the client should start connecting to the ADS and providing LAN-mode service.

The `mac_addr` and `hw_id` must be set in the `ada_conf` first.

This call returns `0` on success, or `-1` if the client is not enabled or not configured.

ada_client_down()

```
void ada_client_down(void
```

The platform calls `ada_client_down()` when the connection to the local network is lost.

7.2 Property Subsystem

The main purpose of the Ayla Embedded Agent is to transfer datapoint values for properties. The agent does this through handlers called property managers. An application can have several property managers, but usually just has one. Each property manager handles one or more properties.

A property manager registers with the Ayla Embedded Agent, providing a structure containing function pointers to the various entry points. Afterwards, the property manager may call the agent to send a new property value to the ADS. The agent may also call the property manager to get or set a property value.

The agent calls the `connect_status()` function of the property manager whenever reachability to the ADS or LAN clients changes.

Several property interfaces are asynchronous. For example, when a property is requested by the agent, the response may be delivered immediately or at a later time, for example, after an interaction with an external device completes.

To send a new value for a property, the property manager calls the `ada_prop_mgr_send()`. This call does not send the property immediately, but merely indicates that the property manager has something to send. When the Ayla Embedded Agent sends the data, the agent then calls the `send_done()` function of the property manager to report success or failure.

When receiving a property value from the ADS, the agent calls the `prop_recv()` function of the property manager.

Following describe three important types of information about the Property subsystem.

enum ayla_tlv_type

This enumerated type indicates the type of a property value. Only some of the possible values are used for property types. Others have internal uses. The details of these property types are described in the *Ayla Module and MCU Interface Specification*. Table 4 provides the property types and their meaning.

Table 4: Property Types

Type	Description
ATLV_INT	Integer, 32-bit
ATLV_UINT	Unsigned integer, 32-bits
ATLV_BIN	Binary bytes, up to 255 bytes long
ATLV_UTF8	UTF-8 encoded character string (see note below)
ATLV_BOOL	Boolean value (0 or 1)
ATLV_CENTS	A decimal number multiplied by 100 in a 32-bit integer
ATLV_SCHED	A schedule describing actions to be taken at specified times
ATLV_LOC	A file property location

NOTE UTF-8 string values may be up to 1024 bytes long, not counting the NUL termination. The string must still be no longer than 1024 bytes after the agent applies escapes required for JSON strings. For example, a new-line, carriage-return, tab, backslash, or double-quote takes 2 bytes to JSON-encode. A byte value between 1 and 0x1f takes 6 bytes to JSON-encode, as does any invalid UTF-8 byte.

Destination Mask

Several APIs use a parameter or variable called the destination mask. This is an unsigned 8-bit integer value in which each bit represents a destination, and may indicate that a property should be sent to that destination, or that connectivity is available to that destination. Table 5 shows an example of this.

Table 5: Example of Destination Mask

Integer Value	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Destination	Schedule	LAN 6	LAN 5	LAN 4	LAN 3	LAN 2	LAN 1	ADS

Error Codes

Several APIs return or use `enum ada_err`. The `enum ada_err` value provides the reason for failure so that an appropriate action may be taken. Error values from `enum ada_err` are always negative and can therefore be used with an interface that returns a zero or positive value on success. Zero generally indicates success; however, some interfaces return a Boolean result of `0` or `1`, neither of which is an error. Table 6 provides the error codes and a short description.

Table 6: Error Codes

Error Code	Description
AE_OK	Success (zero).
AE_IN_PROGRESS	Successfully started, but not finished.
AE_ALLOC	Memory allocation failed.
AE_ERR	Non-specific error used when nothing else is appropriate.
AE_NOT_FOUND	Property or other object not found.
AE_INVAL_VAL	Invalid value for the property or other object.
AE_INVAL_TYPE	Incorrect type for the property or other object.
AE_BUF	Network buffer shortage.
AE_BUSY	Another operation is in progress
AE_LEN	Invalid or excessive length.
AE_INVAL_STATE	Invalid state, the API was called without correct prerequisites.
AE_TIMEOUT	Operation exceeded a time limit.

Error Code	Description
AE_ABRT	Connection aborted.
AE_RST	Connection reset.
AE_CLSD	Connection closed.
AE_NOTCON	Not connected.
AE_INVAL_NAME	Invalid property name.
AE_RDONLY	Ayla Embedded Agent tried to set a from-device property.
AE_CERT_EXP	TLS certificate is not yet valid or has expired.
AE_INVAL_OFF	Invalid file offset.
AE_FILE	Generic file error during file transfers.

7.3 Alternative Property Subsystems

This section describes two alternative property interfaces that can be used by embedded applications:

- Simple Table Property Subsystem. This is the easier of these two alternative property interfaces. This subsystem is a way to handle properties in a synchronous manner in cases where the property manager can respond immediately (synchronously) to property requests.
- Primitive Property Subsystem. This is a more general interface of use in asynchronous environments that might require a message to be sent and a response to be received before responding to property requests.

Sending a property value from the device to the cloud is always asynchronous. The request could take some amount of time and the property interfaces allow for that and provide a way of indicating some time later that the action succeeded or failed. The Simple Property Subsystem manages this by remembering which properties need to be sent and sending them when it can.

7.3.1 Simple Property Subsystem

The Simple Property Table Subsystem handles properties using tables that list each property by name.

The table is an array of structures `ada_sprop`. Each entry has a property name and type, a pointer to the variable holding the value, and a pointer to a function that is called to handle changes to the value.

This section describes the following important aspects of the Simple Property Table Subsystem:

- `struct ada_sprop`

struct ada_sprop

The structure `ada_sprop` and related definitions are in the header file <`ada/sprop.h`>. The definition of that structure is as follows:

```
struct ada_sprop {
      const char *name;
      enum ada_tlv_type type;
      void *val;
      size_t val_len;
      ssize_t (*get)(struct ada_sprop *, void *buf, size_t len);
      enum ada_err (*set)(struct ada_sprop *, const void *buf, size_t len);
};
```

Table 7 provides descriptions of the structure `ada_sprop`.

Table 7: Structure `ada_sprop`

Struct `ada_sprop`	Description
`name`	The `name` member is the name of the property. This should match a name used in the Ayla service template for the device.
`type`	The "type" enum gives the type of the property as returned and expected by the get and set functions.
`val`	This is the pointer to the storage for the value. Integer values are placed there as 32-bit binary values in the host byte order.
`val_len`	This is the size of the value in bytes. These are used only by the get or set functions provided here, so it can actually be anything.
`get`	The `get` member points to a function that is called whenever the property value is needed by the ADS or a mobile application. It should store the value into the provided buffer. The len argument specifies the size of the buffer, which should be large enough to receive the value. The `get` function should return the length that was stored or an enum `ada_err` if the get could not be performed for some reason.
`set`	The `set` member points to a function to be called whenever the property value is to be changed by the ADS or a mobile application or a schedule. The function may fetch the value from the provided buffer.

Library-provided Handlers

The ADA library provides functions that can be used directly in the property table or by application handlers that are called from the property table. These functions make it easier to handle simple properties. These functions match the `struct ada_sprop` members for `get` and `set` and are:

```
ssize_t ada_sprop_get_int(struct ada_sprop *, void *, size_t)
ssize_t ada_sprop_get_uint(struct ada_sprop *, void *, size_t)
ssize_t ada_sprop_get_bool(struct ada_sprop *, void *, size_t)
ssize_t ada_sprop_get_string(struct ada_sprop *, void *, size_t)

enum ada_err ada_sprop_set_int(struct ada_sprop *,
                        const void *, size_t)
enum ada_err ada_sprop_set_uint(struct ada_sprop *,
                        const void *, size_t)
enum ada_err ada_sprop_set_bool(struct ada_sprop *,
                        const void *, size_t)
enum ada_err ada_sprop_set_string(struct ada_sprop *,
                        const void *, size_t)
```

Each `get` or `set` routine uses the `val` and `val_len` fields of the supplied property table entry to get or set the value and place it in the supplied buffer.

If you want the application to do additional checks or operations on the value, the application may do this by supplying its own `set` routine in the table. It may be convenient to call the library-supplied handler from that routine in order to perform size conversions and type checks, and then to reflect the newly set value onto an output pin or LED, for example.

Creating a Simple Property Subsystem

To add a table of properties to the Ayla Embedded Agent, use the function `ada_sprop_mgr_register()` as follows:

```
enum ada_err ada_sprop_mgr_register(char *name, struct ada_sprop *table,
unsigned int entries)
```

The `name` argument gives the group of properties a name for debugging messages.

The `table` argument is the address of the property table. The third argument is the number of entries in the table. This returns `0` on success. The only possible error is a memory allocation failure (`AE_MEM`).

Sending a Simple Property

To send a property to the cloud and LAN-mode clients, use `ada_sprop_send()` as follows:

```
enum ada_err ada_sprop_send(struct ada_sprop *sprop)
```

This sends the value of the property pointed to by `sprop`, which should point to the entry in the table that was used in `ada_sprop_mgr_register()`. The `get()` function in `sprop` is used to get the current value, and that value is saved until it is sent to the service. There is no output from the application if the value is not sent successfully.

Sending a Simple Property by Name

To send a property to the cloud and LAN-mode clients, specifying the property name, use `ada_sprop_send_by_name()` as follows:

```
enum ada_err ada_sprop_send_by_name(const char *name)
```

This sends the value of the property specified by name, The `get()` function in `sprop` is used to get the current value, and that value is saved until it is sent to the service.

This function returns `0` if the request is started successfully. There is no feedback to the application if the value is not sent successfully.

File Property Upload and Download

A file property is required to upload or download datapoints of large sizes. The library requires multiple packets to send or fetch binary data during a file transfer. In case of network connection loss, packets can be re-transmitted.

The library requires a file property to be of `ATLV_LOC` type and the `void *val` pointer to be of `struct file_dp` type. This is required to keep track of the current status of the ongoing file transfer. There can only be one outstanding file transfer active at a time. The `file_dp` structure looks as follows:

```
struct file_dp {
  enum file_dp_state state;
  struct ada_sprop *sprop;
  u32 next_off;
  u32 tot_len;
  size_t chunk_size;
  u8 aborted;
  void *val_buf;
  char loc[PROP_LOC_LEN];
  size_t (*file_get)(struct ada_sprop *sprop, size_t off,
      void *buf, size_t len);
  enum ada_err (*file_set)(struct ada_sprop *sprop, size_t off,
      void *buf, size_t len, u8 eof);
}
```

In the above structure, `file_dp_state` enumeration can have the following values -

```
enum file_dp_state {
  FD_IDLE = 0,       /* nothing to do */
  FD_CREATE,  /* send datapoint create */
  FD_SEND,    /* send datapoint value */
  FD_RECV,    /* request and receive datapoint value */
  FD_FETCHED, /* send datapoint fetched opcode */
  FD_ABORT,   /* abort the datapoint operation */
}
```

The structure keeps a pointer to the `sprop` structure of the file property. The `next_off` bytes attribute keeps track of the progress of the file transfer. The `tot_len` bytes attribute is used during file upload to know the file size to be uploaded. The `chunk_size` bytes attribute records the maximum acceptable chunk sizes managed by the application. The default value is set to 512 bytes; for example, the library will send chunks of 512 bytes to the application during download, or the application will send chunks of 512 bytes to the library during upload. The `aborted` attribute indicates if the last file transfer for that file property was successfully aborted or not. The `val_buf` pointer points to a buffer of `chunk_size` bytes during file transfer. The `loc` attribute is used to save the current Ayla Cloud location that can be used to GET a remote URL where the file can be uploaded/downloaded.

The `file_get` callback function should be defined for file uploads. The library uses this callback function to read chunks of the file currently being uploaded. The `file_set` callback function should be defined for file downloads. Library uses this callback function to send chunks of the file currently being downloaded up to the application.

To set the appropriate `file_get` and `file_set` callback functions and the maximum allowed `chunk_size` that can be handled by the application during file transfers use the following API:

```
void ada_sprop_file_init(struct file_dp *dp,
  size_t (*file_get)(struct ada_sprop *sprop, size_t off,
      void *buf, size_t len),
  enum ada_err (*file_set)(struct ada_sprop *sprop, size_t off,
      void *buf, size_t len, u8 eof),
  size_t chunk_size)
```

To start a file upload, use the `ada_sprop_file_start_send` API. For this API, `name` is the name of the file property and `len` is the total size of the file to be transferred.

```
enum ada_err ada_sprop_file_start_send(const char *name, size_t len)
```

Internally, when this API is called, the simple property manager sends a datapoint create request (corresponds to the `FD_CREATE` state) for the file property. Ayla Cloud responds with a location, on which, a `GET` request is initiated to find the remote URL where the file can be uploaded. Once this happens, the simple property subsystem starts retrieving file chunks by calling the file property's `file_get` callback and sends them to the remote URL

(corresponds to the FD_SEND state). When the complete file is uploaded, a PUT is done on the Ayla Cloud location to mark the transfer complete. At this point the library sends a PME_FILE_DONE event to indicate that the upload is complete.

To start a file download, the ada_sprop_file_start_recv API can be used. For this API, name is the name of the file property, buf is the pointer to the Ayla Cloud location that holds the remote URL for the file to be downloaded, len is the length of the buf and off is the offset from which the file needs to be downloaded.

```
enum ada_err ada_sprop_file_start_recv(const char *name, const void
*buf, size_t len, u32 off)
```

Internally, when this API is called, the simple property manager first acquires the remote URL where the file is stored using Ayla Cloud location held by buf. Once this happens, the library starts downloading the file chunks (corresponds to the FD_REQ state). Simple property subsystem uses the file_set callback function to send the chunks to the application. The final chunk has the eof flag set. The library also indicates the completion of file download with a PME_FILE_DONE event. At this point of time, the file needs to be marked fetched by clearing the Ayla Cloud location. Currently, this is already taken care of by the simple property subsystem; however, the application can take charge of it by calling the ada_sprop_file_fetched API (corresponds to the FD_FETCHED state). For this API, name is the name of the file property.

```
enum ada_err ada_sprop_file_fetched(const char *name)
```

At any time, the file transfer can be aborted by using the following API. This API aborts any current ongoing file transaction.

```
void ada_sprop_file_abort(void)
```

This may be helpful in the case when other property updates from the Ayla Cloud or LAN client are received during file transfer. The library indicates this by sending a PME_NOTIFY event.

Available Destinations

The simple property manager exports a variable indicating the currently available destinations; refer to Destination Mask under Property Subsystem (Section 8.2). The variable is ada_sprop_dest_mask:

```
extern u8 ada_sprop_dest_mask
```

Example of Simple Property Usage

This usage example of sprop implements one property, Blue_LED to control an LED.

```
#include <wm_os.h>
#include <wmstdio.h>
#include <board.h>

#include <ada/libada.h>
#include <ada/sprop.h>

static u8 blue_led;
```

```
/*
 * Demo set function for bool properties.
 */
static enum ada_err demo_led_set(struct ada_sprop *sprop,
                                 const void *buf, size_t len)
{
    int ret = ada_sprop_set_bool(sprop, buf, len);
    if (!ret) {
        set_led(blue_led);
    }
    return ret;
}

static struct ada_sprop demo_props[] = {
    { "Blue_LED", ATLV_BOOL, &blue_led, sizeof(blue_led),
        ada_sprop_get_bool, demo_led_set },
};

/*
 * Initialize property manager.
 */
void demo_init(void)
{
    ada_sprop_mgr_register("ledevb", demo_props,
        ARRAY_LEN(demo_props));
}
```

Note the following in this example:

- The function `demo_init()` registers the `demo_props` table as a set of simple properties. The `ARRAY_LEN()` macro evaluates to 1 in this case, the number of entries in the table.
- The function `demo_led_set()` is called by the Ayla Embedded Agent whenever it receives a new value for the `Blue_LED` property. This calls `ada_sprop_set_bool()`, a libada function that handles any Boolean property, and puts its value in the location (`blue_led`) specified in the `demo_props` table entry. If that works, it then calls the `set_led()` function, which the application code provides to turn on or off the desired LED.
- To add a second LED property to this example, create another table entry naming the new property and using the same set function. To distinguish between the LEDs in `demo_led_set()`, compare the `sprop->val` pointer to `& blue_led` or the address of the added LED variable. Alternatively, use a string compare on `sprop->name`.
- The LEDs in this example are to-device (input) properties.
- For a from-device (output) property, the `set()` function pointer in the table can be `NULL`. The `get()` function may have to read a GPIO pin or just a variable that holds the latest state of the property to be sent. To send a property, the application calls `ada_sprop_send_by_name()`, which calls the `get()` function to get the value. The `get()` function can be called anytime that the Ayla Embedded Agent needs the property value, perhaps due to a request from a LAN client.

7.3.2 Primitive Property Subsystem

The Primitive Property Subsystem is the low-level interface to properties. This subsystem is very flexible, but requires the individual property manager to validate property names and determine how to handle the various events. The other property managers are built on top of this set of interfaces.

There can be several individual property managers. When performing a property-specific operation, for example, a set operation, the agent actually calls the appropriate handler in ***all*** registered property managers until one claims the property by returning success or an error other than `AE_NOT_FOUND`. This means that each property manager should check the name first and if it is not a property the property manager handles, return `AE_NOT_FOUND`.

If a property manager wants to handle all properties, the property manager should be registered after any others. This might be useful when sending updates to an external processor without knowing which properties that external processor handles.

This section describes the following important aspects of the Primitive Property Subsystem:

- struct prop
- ada_prop_mgr_register()
- ada_prop_mgr_ready()
- struct prop_mgr
- ada_prop_mgr_request()
- ada_prop_mgr_send()
- ada_prop_mgr_get()
- ada_prop_mgr_set()
- ada_prop_mgr_dp_put()
- ada_prop_mgr_dp_get()
- ada_prop_mgr_dp_fetched()
- ada_prop_mgr_abort()

struct prop

The prop structure is used in several APIs. This represents a property and a datapoint value in the application and contains the following fields:

- `const char *name`

  This field contains the property name.

- `void *val`

  This field points to the value of the property, for example, a string, 32-bit integer, or a byte, depending on the type.

- `size_t val_len`

The `val_len` member contains the maximum length of the value. For integers, this field is the actual length. For strings (UTF-8), this field contains the maximum length in bytes, including NUL termination.

- `enum ayla_tlv_type type`

  This field gives the type of the value.

- `void (*prop_mgr_done)(struct prop *)`

  This field provides a function to be called after the property manager is done with the property operation.

ada_prop_mgr_register()

```
void ada_prop_mgr_register(const struct ada_prop_mgr *)
```

This call adds the given property manager to an internal list of property managers. The call also declares that the property manager may handle some set of properties and should be called for property operations as defined in the struct `ada_prop_mgr`. The application should eventually call `ada_prop_mgr_ready()` (described next) as well.

ada_prop_mgr_ready()

```
void ada_prop_mgr_ready(const struct ada_prop_mgr *)
```

This call indicates to Ayla Embedded Agent that the property manager is ready to receive property values. This must be called after `ada_prop_mgr_register()` and after each time the ADS destination becomes available. `ada_prop_mgr_ready()` can be called by the `connect_status()` handler and may be safely called more often than necessary.

The Ayla Embedded Agent does not fetch commands or property values from the ADS until all property managers have indicated their readiness by calling `ada_prop_mgr_ready()`. If properties have changed while connectivity to the ADS was lost, it is desirable to send the new values to the service before calling `ada_prop_mgr_ready()` so that the datapoint values from the device have precedence over any datapoints that may have been created on the ADS during the connectivity loss.

struct prop_mgr

The `prop_mgr` structure (supplied by each property manager) provides a set of pointers to functions that the Ayla Embedded Agent may call to deliver and request property values. The structure is defined as follows:

```
struct prop_mgr {
      const char *name;
       enum ada_err (*prop_recv)(const char *name,
             enum ayla_tlv_type type, const void *val, size_t len,
           size_t *offset, u8 src, void *cb_arg);

        /*
         * Callback to report success/failure of property post to
          * ADS/apps.
         * status will be PROP_CB_DONE on success.
         * fail_mask contains a bitmask of failed destinations.
         */
        void (*send_done)(enum prop_cb_status, u8 fail_mask,
            void *cb_arg);

        /*
         * ADS/app wants to fetch a value of a property.
         * prop_mgr must call (*get_cb) with the value of the property.
         */
        enum ada_err (*get_val)(const char *name,
            enum ada_err (*get_cb)(struct prop *, void *arg,
             enum ada_err), void *arg);

        /*
         * ADC reports a change in it's connectivity.
          * Bit 0 is ADS, others are mobile clients.
         */
        void (*connect_status)(u8 mask);

        /*
         * ADC reports an event to all property managers.
         */
        void (*event)(enum prop_mgr_event, const void *arg);
};
```

Following is a description of each field in this structure.

NOTE These functions may not block waiting for other threads or I/O, except on mutexes.

- `name`

  `const char *name`

  The member `name` points to the name of the property manager for use in debug messages. This member should be a short, readable string.

- `prop_recv`

  ```
  enum ada_err (*prop_recv)(const char *name, enum ayla_tlv_type type,
                    const void *val, size_t len, size_t *offset,
                    u8 source_mask, void *cb_arg)
  ```

  The agent calls `prop_recv()` in the property manager when a new value for a data point has arrived. The `name` argument points to the name of the property.

The `type` argument is the datapoint type as received from the ADS or the LAN client. The `val` argument points to the datapoint value in a format appropriate to the type. The `len` argument gives the size of the value in bytes.

The argument `offset` is used only when handling file properties. This points to the offset of the data being received in bytes from the beginning of the value. The `prop_recv()` handler must advance the pointed-to offset by the number of bytes actually used. Simple property managers do not need to use this at all.

The `source_mask` is a destination mask indicating which destination originated the property update.

The final argument, `cb_arg`, is an opaque callback argument. This is `NULL` except when the `prop_recv()` call is in response to a request from the application to the cloud for a property using `ada_prop_mgr_request()`. For now, the application can ignore this argument.

- `send_done()`

```
void (*send_done)(enum prop_cb_status status, u8 fail_mask, void *cb_arg)
```

The agent calls `send_done()` when a request to send a property has been completed.

On normal completion, the status is `PROP_CB_DONE`. The argument `fail_mask` is set to a mask of destinations to which the datapoint could not be delivered. The argument `cb_arg` has the value passed as `cb_arg to ada_prop_mgr_send()`.

- `get_val()`

```
enum ada_err (*get_val)(const char *name,
                  void (*send_cb)(struct prop *, void *arg, int error),
                  void *arg)
```

The agent calls `get_val()` in the property manager to request the value of a property. The argument `name` gives the name of the property. The argument `send_cb` is a function pointer to the function that should be called back with the value of the property. This callback is called some time later if `get_val()` returns `AE_IN_PROGRESS`. The final argument, `arg`, is a value that must be passed as `arg` to `send_cb()`.

- `connect_status()`

```
void (*connect_status)(u8 mask)
```

The agent calls `connect_status()` whenever the connection to the service or a LAN client is established or broken. The `connect_status()` handler is optional and may be `NULL`.

The `mask` argument is the new value of the available destination mask; refer to Destination Mask under Property Subsystem (Section 8.2). Also refer to the information on ada_prop_mgr_ready() earlier in this section.

- `event()`

  ```
  void (*event)(enum prop_mgr_event event, const void *arg)
  ```

  The agent calls `event()` to indicate the occurrence of certain events that may be of interest to property managers. The `event()` handler is optional and may be `NULL`.

  The argument `event` gives the event type, which may be one of the following:

  - `PME_TIME`

    This indicates the time parameters (time, time zone, daylight savings, and so on) have changed. This is of particular interest to property managers that handle schedules or that merely want to display the correct time.

  - `PME_PROP_SET`

    This indicates that a property was changed when received from or sent to the ADS successfully. The argument `arg` is the property name string. This can be used to trigger an action after a change in any property, whether that property is from the same property manager or not.

  - `PME_NOTIFY`

    This indicates that an ANS notification was received, and some properties may have new values in the cloud, or there may be a command pending for the device in the cloud. This event may be ignored, or the property manager may wish to pause a long-running activity, such as a file property download in order to allow the agent to fetch new property values so that the device can remain responsive to service changes.

  - `PME_ECHO_FAIL`

    This indicates that a property value set by one destination could not be echoed to all other destinations. In this case, the `event arg` parameter points to the name of the property. The property manager application normally wants to set a flag to send the property to the ADS once connectivity is restored.

  - `PME_TIMEOUT`

    This indicates that a timeout occurred while waiting for the property manager to respond to a request. The request is abandoned.

  - `PME_FILE_DONE`

    This indicates that the ongoing file property transaction is complete.

ada_prop_mgr_request()

```
enum ada_err ada_prop_mgr_request(const char *name);
```

This function requests that the Ayla Embedded Agent fetch the value of a property from the ADS. The `name` argument specifies the property name. If `name` is `NULL`, a request for all to-device (input) properties is made.

On success (which is indicated by `0` return value), this starts the request which completes, if possible, sometime later. Failures are not reflected back to the application, and may

include loss of connectivity or a property name that is not in the device template of the ADS.

ada_prop_mgr_send()

```
enum ada_err ada_prop_mgr_send(const struct prop_mgr *mgr, struct prop *prop,
                  u8 dests, void *cb_arg)
```

This call initiates a request to send a property to the specified destinations. The first argument is the property manager that handles the property. The second `arg` is the `prop` structure, which must be available until `(*send_cb)()` is called, indicating that the value has been completely sent.

The `dests` argument indicates which destinations should receive the property.

The opaque `cb_arg` pointer is provided to the `send_cb()` callback.

On success, this function returns `AE_IN_PROGRESS`, indicating the operation was started successfully. If the `dests` mask is `0`, it returns `AE_INVAL_STATE`.

ada_prop_mgr_get()

```
enm ada_err ada_prop_mgr_get(const char name,
      enum ada_err (*get_cb)(struct prop *prop, void *arg, enum ada_err), void
      *cb_arg)
```

This function requests that a property be fetched from the ADS. The supplied `get_cb()` function is called on completion. The `name` argument gives the property name. The `cb_arg` argument is passed as `arg` to the `get_cb()` function.

ada_prop_mgr_set()

```
enum ada_err ada_prop_mgr_set(const char *name, enum ayla_tlv_type type,
                      const void *val, size_t val_len,
                      size_t *offset, u8 src, void *set_arg)
```

This function requests that a property on the device is set to the value specified. This is usually used only internally in Ayla Embedded Agent, but may be useful for a property manager to set a property handled by another property manager. This function calls the `prop_recv()` handler in the appropriate property manager. See the prop_mgr prop_recv() function for a description of the arguments.

ada_prop_mgr_dp_put()

```
enum ada_err ada_prop_mgr_dp_put(const struct prop_mgr *pm,
                    struct prop *prop, u32 off, size_t tot_len, u8 eof,
                    void *cb_arg)
```

This function is used to queue a file chunk to be uploaded to the remote URL received from the Ayla Cloud location. `tot_len` is used when `off` is `0`. The last chunk should have `eof` set to 1. `cb_arg` is sent back to the `prop_send_done` callback function.

ada_prop_mgr_dp_get()

```
enum ada_err ada_prop_mgr_dp_get(const struct prop_mgr *pm,
                    struct prop *prop, const char *location, u32 off,
                    size_t max_chunk_size,
                    enum ada_err (*prop_mgr_dp_process_cb)(const char
                    *location, u32 off, void *buf, size_t len, u8 eof),
                    void *cb_arg)
```

This function is used to queue the initiation of a download of a file property. The function retrieves the remote URL where the file resides, using the Ayla Cloud `location`. `off` to indicate the offset from which the file is to be downloaded. `max_chunk_size` indicates the maximum allowed chunk size that can be processed by the application at a time. Invoking the `prop_mgr_dp_process_cb` callback function processes these chunks. `cb_arg` is sent back to the `prop_send_done` callback function.

ada_prop_mgr_dp_fetched()

```
enum ada_err ada_prop_mgr_dp_fetched(const struct prop_mgr *pm,
                    struct prop *prop, const char *location,
                    void *cb_arg)
```

This function is used to mark the downloaded file property fetched, and clearing the Ayla Cloud `location`. `cb_arg` is sent back to the `prop_send_done` callback function.

ada_prop_mgr_dp_abort()

```
void ada_prop_mgr_dp_abort(const char *location)
```

This function is used to abort the ongoing file upload/download at the Ayla Cloud `location`.

7.4 Schedules

Each schedule is like a property and its value is a list of time-based conditions and actions to take when those conditions are met. These conditions and actions are expressed as binary TLV items, and documented in the *Ayla Module and MCU Interface Specification*.

The names of the schedules have the same restrictions as property names and must not be the same as any other property name. The schedule feature is implemented on top of the primitive property manager subsystem.

In order for schedules to operate, the platform must have the current UTC time and, for schedules using local time, the local time offset and daylight savings information. These are provided by the Ayla Device Service, but must be maintained by the platform during restarts or power-downs using a hardware RTC or non-volatile memory. Also kept in NVRAM is the time that the schedules were last evaluated to save some work at startup. APIs for these purposes must be provided by the platform.

NOTE Also refer to the document for *Ayla Schedules* (AY006USD3-1) on support.aylanetworks.com for basic information.

The following interfaces for schedules are described in this section:

- ada_sched_init()
- ada_sched_set_name()
- ada_sched_get_index()
- ada_sched_set()
- ada_sched_set_index()
- ada_sched_enable

ada_sched_init()

```
enum ada_err ada_sched_init(unsigned int count)
```

This interface initializes the schedule system. The interface sets the number of schedules to `count` and allocates resources accordingly. The return is `0` on success. This interface initializes the schedule system and sets the number of schedules to `count` and allocates resources accordingly. The return is `0` on success.

ada_sched_set_name()

```
enum ada_err ada_sched_set_name(unsigned int index, const char *name)
```

This interface sets the name of the schedule specified by `index`, which must be less than the count supplied to `ada_sched_init()`. The supplied `name` must be a valid property name.

ada_sched_get_index()

```
enum ada_err ada_sched_get_index(unsigned int index, char **name,
                                 const void *buf, size_t *lenp)
```

This interface gets the name and the binary value of the schedule specified by index in the supplied buffer. This may be used when persisting the schedule value to persistent storage. The `name` argument points to a string pointer that is set by the function. The `buf` argument points to the buffer to receive the value. The `lenp` argument points to a length variable that is initially set to the size of the buffer. On success, this function sets the length variable to the actual length stored and returns `0`.

ada_sched_set()

```
enum ada_err ada_sched_set(const char *name, const void *buf, size_t len)
```

This interface sets the value of the schedule specified by `name` to the contents of the supplied buffer.

ada_sched_set_index()

```
enum ada_err ada_sched_set_index(unsigned int index,
                                 const void *buf, size_t len)
```

This interface sets the value of the schedule specified by index from the contents of the supplied buffer. This is used when loading the schedule values from persistent storage shortly after boot.

ada_sched_enable

```
enum ada_err ada_sched_enable(void)
```

This interface enables the scheduling system. It should be called after all schedules have had their names set and saved values restored.

7.5 Logging

Provided that the targeted SDK has a console logging subsystem (a serial console), the Ayla Embedded Agent messages are sent to that log. The Ayla Embedded Agent log messages are classified by subsystem and severity. These are defined in `<ayla/log.h>` and `<ayla/mod_log.h>`.

Subsystems of interest include `mod`, `client`, `notify`, `server`, `log-client`, and `sched`. Others are defined, but not used by the Ayla Embedded Agent. Subsystem `mod` is a default subsystem for module logging; almost all messages are classified as belonging to one of the other subsystems.

The severities used are `info`, `warn`, `err`, `debug`, and `debug2`. The `info`, `warn`, and `err`, messages are enabled by default. Debug gives more details about activity; much of it is of interest only to developers and is cryptic. Debug2 gives even more information, including network I/O packets in some cases. Other severities are defined, but not used by the Ayla Embedded Agent.

Following are examples of log messages:

```
[ada] 2015-08-19T17:32:08.500 i m client: ADA-DEMO 1.0-eng
[ada] 17:32:15.739 d m client: http_client_idle_close: session 0
[ada] 17:32:15.744 i m client: ada_client_up: IP 172.16.11.79
[ada] 17:32:15.753 i c client: get DSN AC000W000432408
```

The fields prior to the log message are as follows:

- `[ada]` is the prefix for all Ayla Embedded Agent log messages and may be present or omitted depending on the platform.
- Following the `[ada]` is the UTC time in seconds and milliseconds. When the hour is different from the previous message, or on the first log message, the date is shown as well.
- The UTC is followed by a single character that indicates the severity level of the message, i for info, d for debug or debug2, W for warning, E for error. On multi-threaded platforms this is followed by a short, usually single-letter, indication of the thread that is issuing the log message, that is, m for main thread, c for client.
- The severity level is followed by the subsystem name (for example, client) and then the text of the message. Often, but not always, the text begins with the function name or a portion of it.

Most messages from CLI commands are not considered log messages and do not have this format.

The application may want to generate log messages in the Ayla Embedded Agent format. These should be made under the `mod` subsystem.

The following are descriptions of the interfaces for logging:

log_info()

```
void log_info(const char *format, …)
```

This is a printf-like interface that logs an info message under the default (`mod`) subsystem. This is currently implemented as a macro.

log_put()

```
void log_put(const char *format, …)
```

This interface can be used to log a message of any severity on the default (`mod`) subsystem.

The beginning of the `format` string is a severity designation. These strings are defined as `LOG_INFO`, `LOG_WARN`, `LOG_ERR`, `LOG_DEBUG`, or `LOG_DEBUG2`, and one of these is usually prepended to the format string. For example:

```
log_put(LOG_WARN "%s: error code %d", __func__, err);
```

NOTE Notice the use of string concatenation (no comma after `LOG_WARN`) to prepend the severity designator.

7.6 Over-the-Air (OTA) Updates

The Over-the-Air (OTA) Update feature lets the module accept firmware changes from the service. The Ayla Embedded Agent uses the "host" OTA type, in which the OEM provides the firmware image.

The use of this feature is entirely optional; it is extremely useful, however, and highly recommended. Also, the platform may or may not provide this OTA capability.

It is important that platforms which implement this feature do so in a "brick-proof" manner so that failure never leaves the device in a non-working state.

The platform or application code can provide a different OTA handler for each of the two types of OTAs. The OTA handler provides a set of functions that handle the download and installation of the new firmware.

The ADS and the Ayla Embedded Agent do not place any special requirements on the contents of the OTA firmware image. The OTA image is entirely up to the developer of the platform.

You can start an OTA operation by "deploying" the image either from the Ayla Developers Portal or Ayla OEM Dashboard. Once the agent is notified that an OTA image is going to be deployed, the agent indicates this by calling the OTA handler's `notify()` function. If the image is desired, the handler calls `ada_ota_start()` to prompt the agent to download the image. The agent provides the image to the handler by calling the `save()` function for each part of the image as it arrives. After all of the parts have been given to the handler in sequence, the agent calls the handler's `save_done()` function.

Once the handler has applied the OTA and rebooted to the new firmware image, the handler reports the completion of the operation by calling `ada_ota_report()`.

Following are descriptions of the interfaces used in OTA updates.

- enum patch_state
- ada_ota_register()
- struct ada_ota_ops
- ada_ota_start()
- ada_ota_continue()
- ada_ota_report()

enum patch_state

This enum provides status codes for OTA updates and is the return value for several operations. The ADS understands the codes, and existing assignments must not change. The codes are defined in `<ayla/patch_state.h>`.

Only some of the values apply to the OTA update platform code. Others are used internally. Table 8 describes the codes that may be used by the platform.

Table 8: OTA Update Codes Used by the Platform

Codes	Description
PB_ERR_NEW_CRC	The modified code results in a CRC error.
PB_ERR_STALL	This is not an error. It is an indication that the download should pause until `ada_ota_continue()` is called. This code may be used for temporary resource shortages or a delay imposed by an external system.
PB_ERR_DECOMP	The update had failed to decompress.
PB_ERR_OP_LEN	A patch had a length error.
PB_ERR_FATAL	An unspecified error occurred while applying the update.
PB_ERR_OP	A segment of the update had an invalid opcode.
PB_ERR_STATE	The patch program was in an invalid state.
PB_ERR_CRC	A section of the image being patched had a bad CRC before patching.
PB_ERR_COPIES	More than one block is in the copied state.
PB_ERR_HEAD	The patch file area in flash had a bad structure or length.
PB_ERR_FILE_CRC	The update file had a CRC error.
PB_ERR_ERASE	Persistent storage erasure failed.
PB_ERR_WRITE	Persistent storage write failed.
PB_ERR_SCRATCH_SIZE	The scratch area was too short.
PB_ERR_DIFF_BLKS	The old and new data were in different storage blocks.
PB_ERR_OLD_BLKS	The old data for a section of the patch spanned two blocks.
PB_ERR_NEW_BLKS	The new data for a section of the patch spanned two blocks.
PB_ERR_SCR_ERASE	The scratch block erasure failed.
PB_ERR_SCR_WRITE	The scratch block write failed.
PB_ERR_PROG	An error occurred reading or writing a progress byte.
PB_ERR_PROT	A block to be patched was not in the correct progress state.
PB_ERR_NOFILE	The patch file was not found in persistent storage.

Codes	Description
PB_ERR_HEAD	The patch file could not be read.
PB_ERR_NO_PROG	The update area did not contain a progress area.
PB_ERR_INV_PROG	The progress area was invalid.
PB_ERR_READ	The patch file could not be read.
PB_ERR_DECOMP_INIT	A failure occurred in initializing the decompressor.
PB_ERR_PREV	A previous patch attempt failed.
PB_ERR_OPEN	An error occurred opening the persistent storage device.
PB_ERR_BOOT	The update program did not boot.

ada_ota_register()

```
void ada_ota_register(enum ada_ota_type type, const struct ada_ota_ops *ops)
```

This function registers a set of handlers for the specified `type` of OTA (either `OTA_MODULE` or `OTA_HOST`). To unregister a handler, the second argument may be `NULL`. This function always succeeds or results in an assertion failure if an unrecognized `type` is given. The `ops` structure is described next, `struct ada_ota_ops`.

struct ada_ota_ops

This structure contains pointers to the functions implementing the OTA handler. The `struct ada_ota_ops` is filled in by the handler code and passed to `ada_ota_register()`. This structure must remain valid for the duration of the client operations.

The `struct ada_ota_ops` contains the following function pointers:

- `notify()`

  ```
  enum patch_state (*notify)(unsigned int len, const char *version)
  ```

  The `notify()` function indicates to the handler that an OTA is available. The length and version string are given. If the length or version are unacceptable, `notify()` may return an error. If there is no error, the handler may start the download at any point afterwards by calling `ada_ota_start()`.

- `save()`

  ```
  enum patch_state (*save)(unsigned int offset, const void *buf, size_t len)
  ```

  The `save()` function gives the handler a section of the OTA update to save in persistent storage.

  The first argument gives the offset in the file. It is `0` for the first save in each OTA update.

NOTE Subsequent calls always have an offset that has been advanced by the previous length of the call.

The second argument gives the address of the section of data to be saved.

NOTE The agent may reuse this buffer as soon as `save()` returns.

The third argument gives the length in bytes of the data for this call.

The save function may return `PB_ERR_NONE` on success or another error if the patch could not be stored. `PB_ERR_STALL` indicates that the data was saved, but that `save()` should not be called again until the handler calls `ada_ota_continue()`.

- `save_done()`

  ```
  void (*save_done)(void)
  ```

  The `save_done()` function is called after all sections of the OTA update have been successfully saved. This gives the handler an opportunity to check the image for validity and to boot to the new image. Any problems with the image should be reported using `ada_ota_report()`.

- `status_clear()`

  ```
  void (*status_clear)(void)
  ```

  The `status_clear()` function is called to clear the previously reported status of an OTA operation. This method is optional, and may be left `NULL`.

ada_ota_start()

```
void ada_ota_start(enum ada_ota_type otype)
```

The handler calls the `ada_ota_start` function to start an OTA download. This function may be called from the handler's `notify()` function or later. The first argument gives the type of the OTA to be started.

ada_ota_continue()

```
void ada_ota_continue(void)
```

The handler calls this function to continue a download after a call to `save()` returned `PB_ERR_STALL`.

ada_ota_report()

```
void ada_ota_report(enum ada_ota_status)
```

The handler calls this to the status of an OTA image after it has been completely downloaded. This may be called during the `save_done()` function or at a later time, even in the new image after it has booted.

7.6.1 OTA Considerations

The primary concern around OTA updates is to make sure that the module continues to run either the old image or the new image after the update, even if power is lost at some point during the OTA update.

The SDK may have an OTA mechanism that can be used, including a boot loader that allows installing a new image or two images in flash that can be alternately booted.

Be sure to preserve the configuration during an OTA, and allow for compatibility between old configurations and new firmware images. Some configuration items expected by new images may not be present in the old configuration. These should have reasonable default values.

Often Wi-Fi firmware has to be updated at the same time as an SDK change. Therefore, you may want to combine the Wi-Fi firmware with the rest of the firmware so that they can be updated together.

7.7 CLI Interfaces

All CLI interfaces take as an argument the number of command line arguments and a pointer to an array of the command line argument pointers. They return void. The first argument is always the command name. Following are descriptions of the CLI interfaces:

- ada_conf_oem_cli()
- ada_log_cli()

ada_conf_oem_cli()

```
void ada_conf_oem_cli(int argc, char **argv)
```

This CLI can be used to show the OEM settings and to set the OEM ID, OEM model, and OEM key. The OEM ID and OEM model values entered by the CLI override the compiled-in defaults.

The `ada_conf_oem_cli` interface should be used for manufacturing setup only and may be left out of end-user applications.

Without arguments, the current settings are shown.

With one argument, the OEM ID is set.

If the command is given as `oem model <value>`, the OEM model is set.

With the command syntax `oem key <secret> [<oem-model>]`, the OEM key is set.

The optional OEM-model entered with the key restricts the device to just that OEM model, unless the key is re-entered. If omitted, the current OEM model setting is used. The `oem_model` can be entered as `*` (a single asterisk) to allow the OEM model to be set to anything by the application later.

ada_log_cli()

```
void ada_log_cli(int argc, char **argv)
extern char ada_log_cli_help[];
```

This CLI can be used to show and modify logging settings. The help string may be used if desired to show the usage of the command.

The CLI syntax is:

```
log [--mod <subsystem>] [<level>…]
```

The available subsystems are `client`, `conf`, `dnss`, `mod`, `notify`, `server`, `wifi`, `ssl`, `log-client`, `io`, `sched`, `eth`, and `test`.

If no subsystem is specified, the command affects all subsystems (modules).

Without a level argument, the current logging levels are shown.

The available logging levels are `info`, `warning`, `error`, `debug`, `debug2`, `metric`, `pass`, `fail`, `none`, and `all`. If a logging level is preceded with a hyphen, for example `-debug`, then that level is turned off. Some levels (such as `warning` and `error`) cannot be turned off. Some information messages may occur even when `info messages` are disabled.

8 Required Application Interfaces

This section describes the interfaces that the platform application must implement for use by the Ayla Embedded Agent. These interfaces provide for configuration and logging, among other platform-specific tasks.

8.1 Configuration Subsystem

The platform must provide a method for persistently saving certain configuration items. The device must maintain a unique ID and key that are used to authenticate it with the ADS.

In addition, the ADS sends configuration items, such as the time zone and daylight-savings info, and the LAN key and various modes. These must be persisted and kept across reboot and power-cycles in order for the device to work without Internet connectivity.

8.1.1 Required Configuration Items

Following descriptions of the configuration items:

- id/dev_id
- id/pub_key
- struct ada_conf
- u8 oem[]
- u8 oem_model[]
- adap_conf_sw_build()
- adap_conf_sw_version()
- adap_conf_reg_changed()

id/dev_id

This string is the Ayla Device Serial Number (DSN). It is usually a 15-character value, like AC000W123456789. The device OEM must set this. It can be stored in a one-time-programming area if practical. This is the primary identifier for the device to the Ayla device service.

id/pub_key

This is the 2048-bit RSA public key corresponding to the DSN. The device OEM must set this. This value can be saved in binary or as a base-64 text string. See the description for `ada_pub_key_get()` in Section 9.1.2.

struct ada_conf

The `ada_conf` structure is a global structure that collects several configuration items that control the Ayla Embedded Agent and in particular the client portion of the agent.

There are two sections of fields in the `ada_conf` structure. The first part is composed of fields that are set by the platform application before starting the client. The second part contains items that may be of interest to the platform application. Following are descriptions of the fields in the `ada_conf` structure:

- `u8 enable:1`

  This bit is normally set to `1` during initialization by the application startup. In some situations, you may want to control this bit with a CLI.

- `u8 get_all:1`

  This bit, if set, indicates that the client should fetch all input (to-device) properties upon the first connection to the ADS. This is often desirable, but some applications may want the chance to send properties that the application altered due to LAN mode devices or persisted due to scheduled changes before fetching the latest ADS values.

- `u8 test_connect:1`

  This bit, if set, indicates to the ADS that the connection is for testing at the factory and doesn't indicate the first end-user connection by the device. Test software may want a way to set this temporarily, but it shouldn't be persisted.

- `u8 conf_serv_override`

  This byte, if non-zero, indicates that the client should use the default ADS server name rather than the name that would be implied by the combination of the `oem_model` and `OEM`. This does not override the region and can be generally ignored by the application. This byte can also be set through the cloud and persisted in the configuration.

- `u8 mac_addr[6]`

  The platform must set this field to a unique MAC address for the device before starting the client.

- `const char *hw_id`

  The platform should set this field to a unique hardware ID string of ASCII characters (usually hex digits and dashes). The string should remain valid while the Ayla Embedded Agent is active. This field is sent to the ADS and available as information on the Ayla OEM Dashboard. The field may be used to verify the uniqueness of the DSN assignments.

- `const char *region`

  The two-letter region string indicates the part of the world where the device will operate. This determines the DNS domain that is used for the ADS server. This may be left `NULL` for the default.

  The valid domains are currently:

  - **CN** for the People's Republic of China, which uses ayla.com.cn.
  - **US** (or NULL default) for the rest of the world, which uses aylanetworks.com.

- `const char conf_server[]`

  This field can be used to configure the entire ADS server host name, for example, to connect to a test server. Generally it is left NULL.

- `u16 conf_port`

  This field can be used to set the port used by the server's HTTPS server, for example, to connect to a test server. Generally, the field is set to `0`.

- `u16 poll_interval`

  This field is the number of seconds between polls of the ADS by the ADA client at times when ANS is not reachable. The field is normally is set to 30 seconds by the application. If set to `0`, the client currently uses 5 minutes as the default.

- `char host_symname[]`

  This string is received from the cloud and filled in by the Ayla Embedded Agent. The string holds the symbolic name that comes from the template originally and may be set by the user.

- `char reg_token[]`

  This string is set by the Ayla Embedded Agent and holds the registration token that must be entered by the user to register the device. In the Display Method of registration, the application would need a way to display this string.

NOTE For information on the Ayla registration methods, refer to *Device Onboarding: Ayla Registration Methods (AY006FOR3-1)* on support.aylanetworks.com.

- `u8 reg_user:1`

  This bit is set by the Ayla Embedded Agent when the device is registered to a user.

u8 oem[]

The application must set this global string to the OEM ID assigned by Ayla. This is typically an 8-character ASCII string composed of hex digits.

u8 oem_model[]

The application must set this global string. The OEM model indicates the template to be used. The OEM key is used to authenticate the OEM and OEM model.

adap_conf_sw_build()

```
const char *adap_conf_sw_build(void)
```

This function should return the version string to report to the ADS as the module software version. Among other things, this is used to as information to filter the available module OTA images. The function may return `ada_version_build`, the supplied build info for the Ayla Embedded Agent.

adap_conf_sw_version()

```
const char *adap_conf_sw_version(void)
```

This function should return the module version string to report to LAN clients. The function may return `ada_version`, the supplied version of the Ayla Embedded Agent.

adap_conf_reg_changed()

```
void adap_conf_reg_changed(void)
```

This function must be provided by the application. The function is called when `ada_conf.reg_user` changes. This function may be used to indicate that progress to the user with a display or LED, or may do nothing.

8.1.2 Persisting Configuration

These functions must be provided by the application to save configuration items in and restore them from persistent storage. Following are descriptions these functions.

adap_conf_get

```
int adap_conf_get(const char *name, void *buf, size_t len)
```

This function fetches the specified configuration item if found in persistent storage. The `name` argument is the name of the configuration item. The `buf` argument specifies the buffer address, and the `len` specifies the length in bytes. The function should return the number of bytes read, which is generally a string length, but in some cases is the length of binary data read. On error, the function should return `-1` or a negative error number.

adap_conf_set

```
int adap_conf_set_bin(const char *name, const void *buf, size_t len)
```

This function saves a configuration item, specified by the `name`. The source of the data is the buffer specified by the `buf` and the `len` in bytes. Although in many cases the buffer is a string, the buffer can be binary data and no assumptions about the content should be made. The function returns `0` on success, and `-1` or a negative error number on failure.

adap_conf_pub_key_get

```
int ada_conf_pub_key_read(void *buf, size_t len)
```

This function reads the configured device public key as an ASN-1-encoded binary 2048-bit RSA key. The key must be configured in the platform during manufacturing and may be saved in binary or base-64 or some other format, but delivered in binary. If a name/value configuration system is used, the name `id/pub_key` is recommended.

adap_conf_oem_key_get

```
int ada_conf_oem_key_read(void *buf, size_t len)
```

This function reads the encrypted OEM key in binary. This may have been saved by a CLI command or another method during manufacturing. The arguments `buf` and `len` specify the area to receive the key. The return value is the length of the key read, or a negative value on error. If a name/value configuration system is used, the name `oem/key` is recommended.

8.2 Wi-Fi Subsystem

The Ayla Embedded Agent, Ayla Device Wi-Fi (ADW), and alternative Wi-Fi subsystems require Wi-Fi interfaces. If there is no Wi-Fi interface, these may be stubbed out, but must still be supplied. This section provides descriptions of the following functions for Wi-Fi:

- adap_wifi_features_get()
- adap_wifi_in_ap_mode()
- adap_wifi_get_ssid()
- adap_wifi_stayup()
- adap_wifi_show_hist()
- adap_net_get_signal()

adap_wifi_features_get()

```
enum ada_wifi_features adap_wifi_features_get(void)
```

This function returns a mask of features provided by this version of the ADW. It can depend on compile options and on the version of the ADW being used. The features may include:

`AWF_SIMUL_AP_STA`	Indicates that simultaneous AP and Station mode can be used.
`AWF_WPS`	Indicates that Wi-Fi Protected Setup (WPS) is supported.
`AWF_WPS_APREG`	Indicates that AP-Mode registration of the device to a user during WPS setup is supported.

All of the values that are supported are ORed together. The caller can test for the presence of a given feature using code that tests the associated bit; for example:

```
enum ada_wifi_features features;

features = adap_wifi_features_get();
if (features & AWF_WPS) {
      printf("WPS supported\n");
}
```

adap_wifi_in_ap_mode()

```
int adap_wifi_in_ap_mode()
```

This returns a non-zero value if the AP interface is active, and zero otherwise.

adap_wifi_get_ssid()

```
int adap_wifi_get_ssid(void *buf, size_t len)
```

This function fills in the supplied buffer with the (Service Set Identifier) SSID of the connected network. The `len` argument gives the total length of the buffer available, which must be at least 33 bytes long. The returned value is the length of the SSID in bytes. It will be `0` or a negative number if the station interface is not connected to a network.

adap_wifi_stayup()

```
void adap_wifi_stayup(void)
```

This function instructs the ADW that the interface is in use and connectivity should be maintained. The Ayla Embedded Agent uses this to inform ADW when the internal web server is in use so that AP mode remains active for some unspecified time longer.

adap_wifi_show_hist()

```
void adap_wifi_show_hist(void)
```

This should cause the history of that last several Wi-Fi connection attempts to be logged to the console as Wi-Fi info messages. This function is called before logs are sent to the logging service the first time. This gives the Wi-Fi subsystem a chance to include its status and history in the first connection to the log server.

adap_net_get_signal()

```
int adap_net_get_signal(int *signalp)
```

The Ayla Embedded Agent uses this function to get the current signal strength from the network subsystem. If a wireless network is connected, it should set the integer pointed to by `signalp` to the signal strength in dBm and return `0`. If no wireless network is connected or this functionality is not provided, the return is `-1`.

4250 Burton Drive, Santa Clara, CA 95054
Phone: +1 408 830 9844
Fax: +1 408 716 2621